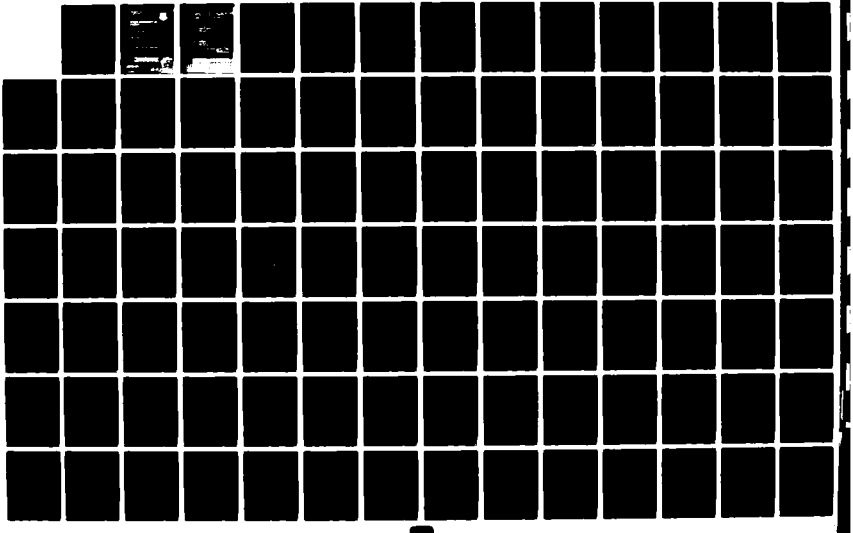
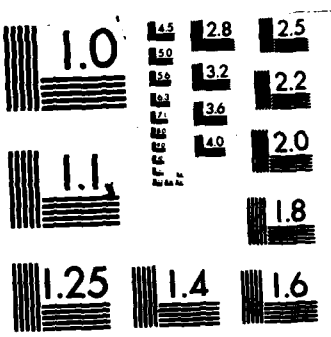
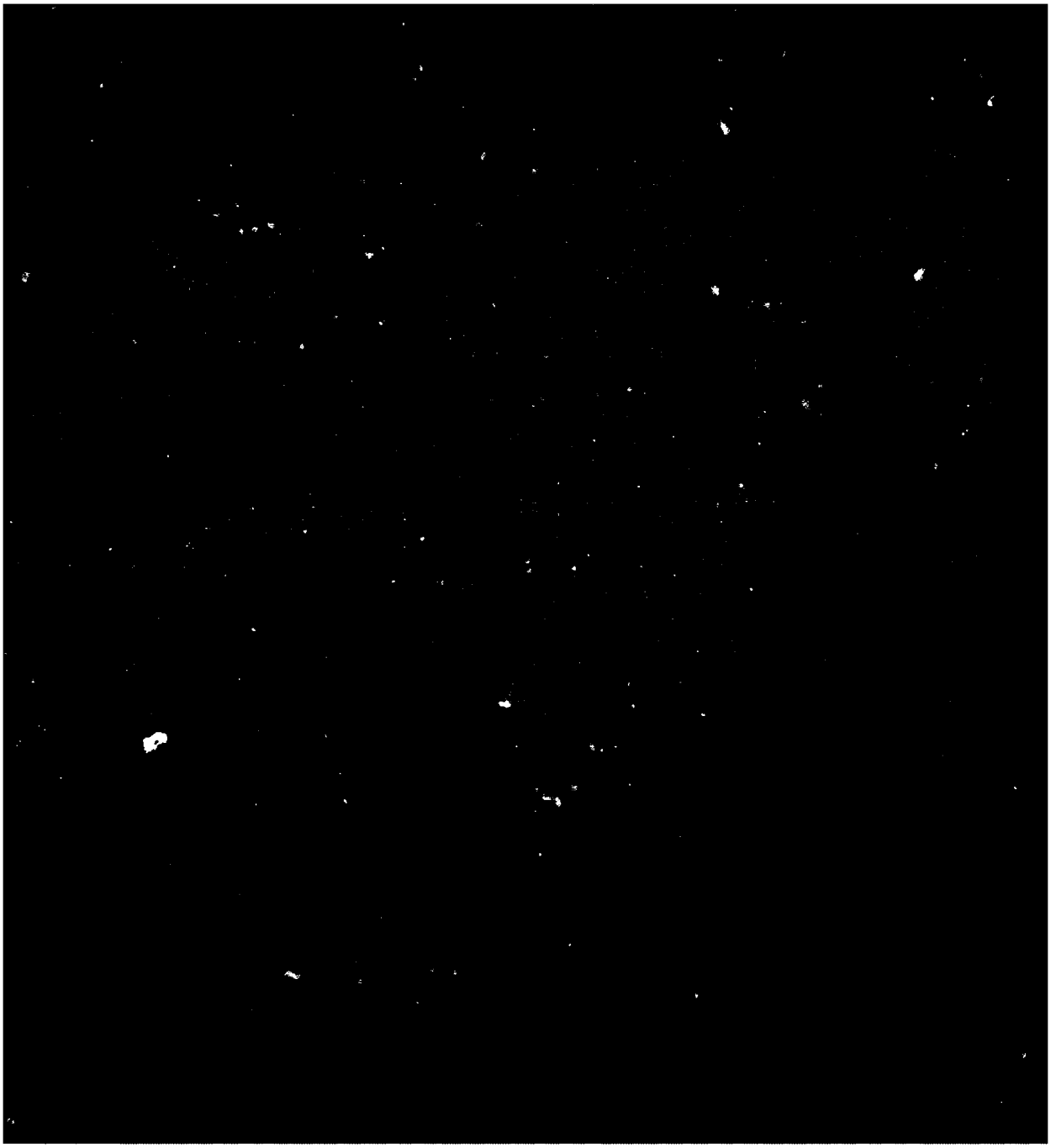
AD-A138 477 SOFTWARE INTEROPERABILITY AND REUSABILITY VOLUME 1(U) 1/2
BOEING AEROSPACE CO SEATTLE WA P E PRESSON ET AL.
JUL 83 RADC-TR-83-174-VOL-1 F30602-80-C-0265

UNCLASSIFIED F/G 9/2 NL







UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-174, Vol I (of two)	2. GOVT ACCESSION NO. AD-A133477	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE INTEROPERABILITY AND REUSABILITY		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Aug 80 - Feb 83
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) P. Edward Presson, Jonathan V. Post Juitien Tsai, Robert Schmidt Thomas P. Bowen		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0265
9. PERFORMING ORGANIZATION NAME AND ADDRESS Boeing Aerospace Company PO Box 3999 Seattle WA 98124		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812019
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE July 1983
		13. NUMBER OF PAGES 176
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Joseph P. Cavano (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Quality
Software Resuability
Software Metrics
Software Measurement
Software Interoperability.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Software metrics (or measurements) which predict software quality were extended from previous research to include two additional quality factors: interoperability and reusability. Aspects of requirements, design, and source language programs which could affect these two quality factors were identified and metrics to measure them were defined. These aspects were identified by theoretical analysis, literature search, interviews with project managers and software engineers, and personal experience. (over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

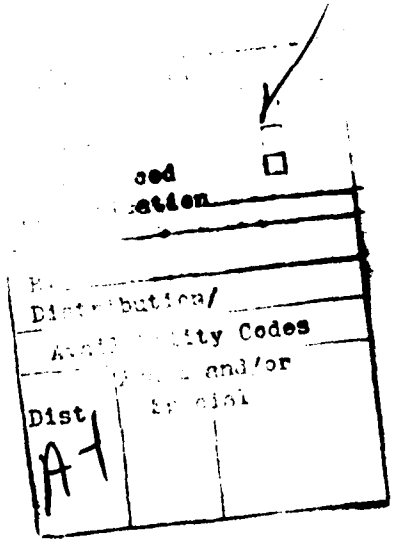
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

GUIDE Book

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

A ~~Guidebook~~ for Software Quality Measurement was produced to assist in setting quality goals, applying metrics and making quality assessments.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

SUMMARY

Software metrics (or measurements) which predict software quality were extended from previous research (RADC Contracts F30602-76-C-0417 and F30602-78-C-0216) to include two additional quality factors: interoperability and reusability. Aspects of requirements, design, and source language programs which could affect these two quality factors were identified and metrics to measure them were defined. These aspects were identified by theoretical analysis, literature search, interviews with project managers and software engineers, and personal experience.

The metrics for each of these two quality factors were structured in a metric framework, and data worksheets were designed to gather the necessary data to validate the metrics. The worksheet data was collected from several projects; from that data metric scores for each software module were computed. Mathematical analyses of the metric scores for each quality factor were then performed.

Because interoperability and reusability were determined to be very different qualities, the frameworks and research that followed diverged. *Interoperability is primarily* a system consideration, whereas reusability is usually focused on the module level.

Validation of interoperability metrics was approached very conservatively; it could only be done at the system level for two reasons. First, interoperability was defined at the system level, not at the module level. Second, interoperability ratings were only available at the system level; there was no way to meaningfully rate the interoperability of a single module. The limited number of software projects with interoperability requirements constrained the use of normal validation techniques, for these techniques require more data points (i.e., metric scores plotted versus interoperability ratings) than there were systems available for study.

Exploratory data analysis was performed on the data to ascertain the meaningfulness of the data. Interoperability ratings for each project were obtained to validate the metric framework by a modified Delphi survey technique. Since these ratings were estimates and therefore subject to a considerable possibility of error, a sensitivity analysis was performed to determine the impact on the solution due to small errors in the interoperability ratings. The results indicated that a multivariate analysis of the relationship between

the metric framework and the interoperability analysis was very sensitive to small errors in the interoperability ratings. So sensitive was this relationship that small changes in the interoperability ratings could completely alter the solution. Thus, no confidence could be given to any solution. However, several metrics were found to be descriptive, and could be used as the basis for further research.

Because interoperability proved to be such a problematical quality to discuss, predict, and measure; extensive interviews with project personnel were undertaken early in the research to aid in construction of a reasonable interoperability metric framework. These series of interviews provided considerable insight into the nature of interoperability which is discussed in the body of this report.

Reusability proved to be more tractable. Reusability ratings were computed at the module level and used to validate the reusability metric scores using regression and correlation analysis techniques. These results were plotted and evaluated by computer using the metric data base collected from four major projects. The most successful approach came from using productivity figures from the projects to estimate the effort spent to reuse software. Good correlation between these reusability quality ratings, or productivity measures, and system reusability metric scores were obtained. Using the system reusability measures, module-level reusability measures were computed and correlated against the module metric scores. The results showed a positive correlation between several metrics and the software system reusability ratings.

Volume I describes the technical effort accomplished under this contract.

Volume II is a guidebook for using the software quality measurement framework.

SOFTWARE INTEROPERABILITY AND REUSABILITY
FINAL REPORT - Volume I

TABLE OF CONTENTS

	Title	Page
SUMMARY		
1.0	INTRODUCTION	1
1.1	Objectives of Research	1
1.2	Background	1
1.3	Software Quality Framework	5
1.4	Technical Approach	8
2.0	TECHNICAL ACCOMPLISHMENTS	11
2.1	Software Interoperability & Reusability	11
2.1.1	Top Level Framework	11
2.1.2	Cost/Benefit Perspective	14
2.1.3	Quality Concerns	15
2.2	Interoperability	17
2.2.1	Key Concepts	18
2.2.2	System Characteristics	19
2.2.3	Interoperability Criteria	40
2.2.4	Tradeoffs Between Interoperability and Other Quality Factors	48
2.2.5	Data Collection	53
2.2.6	Metric Validation	55
2.2.7	Conclusions and Recommendations	69
2.3	Reusability	74
2.3.1	Key Concepts	76
2.3.2	System Characteristics	78
2.3.3	Reusability Criteria	91
2.3.4	Reusability Concepts vs Reusability Criteria	98
2.3.5	Reusability Metrics	102

TABLE OF CONTENTS (Continued)

		Title	Page
	2.3.6	Impacts of Application on Reusability	106
	2.3.7	Reusable Software Development Guidelines	106
	2.3.8	Tradeoff Between Reusability and Other Quality Factors	114
	2.3.9	Data Collection	118
	2.3.10	Conclusions	127
	2.3.11	Recommendations	127
2.4		Impact on AMT	129
2.5		Quality Measurement Manual Enhancement	130
References			
Appendix	A	Project Regression Analysis Results	A-1
	B	Regression Analysis—Combined Projects	B-1
	C	Reusability Metric Plots	C-1
	D	Multivariate Analysis Results	D-1

LIST OF FIGURES

		Page
1.1-1	Software Quality Model	3
1.3-1	Software Quality Framework (Old)	6
1.3-2	Software Quality Model (New)	6
1.4-1	Software Interoperability and Reusability Task Flow	9
2.1-1	Software Quality Framework (Old)	12
2.1-2	Operational Use of the Interoperability and Reusability Quality Factors	16
2.2-1	Cost to Make Interoperable Depends on Functional Overlap and Module Strength	33
2.2-2	Original & New Interoperability Framework	41
2.2-3	Metric Combinations vs. Interoperability Ratings	34
2.3-1	Reusability Hierarchy	75
2.3-2	Reusability Framework	94
2.3-3	Software Development Phases	109
2.3-4	Productivity vs. Reusability Factor Metric	121
C-1	Reusability Rating vs. Metric FS.1	C-2
C-2	Reusability Rating vs. Metric GE.2	C-3
C-3	Reusability Rating vs. Metric ZD.2	C-4
C-4	Reusability Rating vs. Metric MO.2	C-5
C-5	Reusability Rating vs. Metric SC.1	C-6
C-6	Reusability Rating vs. Metric SC.2	C-7
C-7	Reusability Rating vs. Metric SC.4	C-8
C-8	Reusability Rating vs. Metric SD.1	C-9
C-9	Reusability Rating vs. Metric SD.3	C-10
C-10	Reusability Rating vs. Metric SI.1	C-11
C-11	Reusability Rating vs. Metric SI.3	C-12
C-12	Reusability Rating vs. Metric SI.4	C-13

LIST OF TABLES

		Page
1.1-1	Enhanced Software Quality Measurement Framework	7
2.1-1	Classification of Quality Factors	13
2.2-1	Characteristics that Impact Software Interoperability	20
2.2-2	Interoperability Concepts vs. System Concepts	31
2.2-3	Seven Criteria Determine Functional Strength	34
2.2-4	Interoperability Tradeoffs with Other Quality Factors	52
2.2-5	Interoperability Metric Summary (by Project)	58
2.2-6	Interoperability Metric Summary (by Criteria)	59
2.2-7	Interoperability Metric Summary	59
2.2-8	Median Metric Scores by Project	60
2.2-9	Mean Cumulative Worksheet Scores	65
2.2-10	Last Sample Metric Scores	67
2.3-1	Key Concepts of Software Reusability	76
2.3-2	System Characteristics for Reusability	78
2.3-3	Impact on Reusability by Characteristics	90
2.3-4	System Characteristics and Reusability Criteria	92
2.3-5	Definition of Reusability Criteria	93
2.3-6	Reusability Concepts vs. Reusability Criteria	99
2.3-7	Software Reusability Quality Metrics	103
2.3-8	General Design Guidelines for Reusability	108
2.3-9	Tradeoff of Reusability with Other Quality Factor	114
2.3-10	Reusability Criteria and Other Quality Factor	115
2.3-11	Worksheet Data Collected	119
2.3-12	Metrics Summary (By Criteria)	120
2.3-13	Productivity vs. Reusability Metrics	120
2.3-14	Summary of Module Reusability Rating Values	123
2.3-15	Regression Analysis Example	124
2.3-16	Reusability Metrics with Correlation	125

LIST OF TABLES (Continued)

		Page
A-1	Project 1 - Regression Analysis Summary	A-2
A-2	Project 2 - Regression Analysis Summary	A-4
A-3	Project 3 - Regression Analysis Summary	A-6
A-4	Project 4 - Regression Analysis Summary	A-9
B-1	Projects - Combined Regression Analysis Summary	B-2
D-1	Multivariate Analysis Summary	D-1

SECTION I
INTRODUCTION

1.1 OBJECTIVES OF RESEARCH

This work was performed under a research contract (F30602-80-C-0265) for Rome Air Development Center, Griffiss AFB, NY. The object of this effort was to develop techniques which can be used to measure software interoperability and reusability. This study looked at the software from the life cycle viewpoint, and examined the software characteristics which contribute to these two software quality factors. The characteristics of the requirements, design, and the actual software source program were considered. This study was conducted to develop and validate proposed metrics for software interoperability and reusability. This effort was also conducted to expand and refine the software quality measurement framework defined in prior Government (RADC) contracted research, Factors in Software Quality, F30602-76-C-0417 and Software Quality Metrics Enhancement, F30602-78-C-0216. The Software Quality Measurement Manual (RADC-TR-80-109, Volume II) which was developed under the latter contract, was modified and expanded to include the new metrics and is included as Volume II of this report.

The results of this study are expected to enhance the methodology for AF software acquisition managers to determine software quality requirements over a project life cycle in terms of its software quality factors and to specify or describe those requirements to contractors. The results are also expected to provide the methodology for software development managers to control the quality of software products using the increased visibility provided by predicting and measuring software quality factors.

1.2 BACKGROUND

Rome Air Development Center (RADC) has been pursuing a program since 1976 to better specify and control software quality. This program has been seeking to identify the key issues and provide a valid methodology for specifying and measuring software quality of major AF weapon systems.

In 1976 RADC and Electronic Systems Division (ESD) sponsored an effort which defined a set of eleven user-oriented characteristics or quality factors (correctness, efficiency, integrity, usability, testability, flexibility, reusability, maintainability, reliability, portability, and interoperability) which extended throughout the software life-cycle. this effort established a hierarchical software quality measurement framework as shown in Figure 1.1-1. The user-oriented factors, designed for use by acquisition managers to specify quality requirements, are at the top level. The software oriented criteria (attributes which indicate quality) and software metrics (quantitative measures of attributes) are at the second and third level, respectively. The metrics represent the most detailed level of the framework. Taken collectively, this hierarchy forms the basis of a model for predicting and controlling software quality. This research was performed under contract F30602-76-C-0147, and was described in technical report RADC-TR-77-369, Factors in Software Quality.

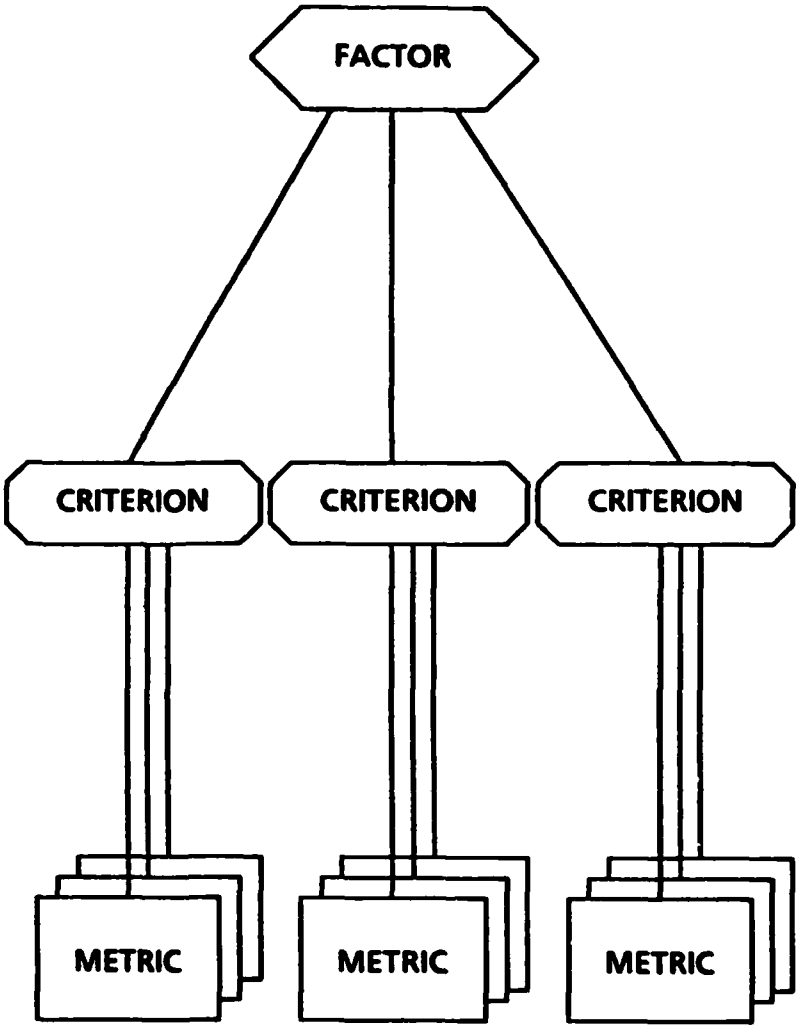


Figure 1.1-1 **Software Quality Model**

In 1978, RADC and the US Army Computer Systems Command sponsored additional research to enhance this framework under contract F30602-78-C-0216, Software Quality Metrics Enhancement. The results of this effort were reported in technical report RADC-TR-80-109, Volume I Software Quality Metrics Enhancement and Volume II Software Quality Measurement Manual. The manual provides methodology to assist the AF acquisition manager in describing to a contractor what quality factors the manager considers the most important.

In 1979, RADC and the US Army Computer Systems Command issued contract F30602-79-C-0267 to develop an Automated Quality Measurement Tool for the H6180/GSOS Computer System. The purpose of this tools was to automate the collection of specific metric data and provide quality measurement assessments. This tool was delivered to the Air Force in September 1981.

In 1980, RADC sponsored further research into the software factors interoperability and reusability with this contract (F30602-80-C-0265). The objective was to enhance the Software Metric Model by incorporating new findings for these factors which had not been extensively studied in prior research contracts. This research was to formulate and validate metrics for interoperability and reusability and to provide this information in a form useful to the AF software acquisition manager.

This contract reflects the increasing importance of these two quality factors in the cost of large military systems with embedded software. As an increasing number of sophisticated systems are deployed, it becomes more important for these systems to be able to transmit, receive, and use information of mutual interest in order to be able to deal effectively with rapidly changing threat environments. Thus, a reliable method for predicting and measuring interoperability is needed to minimize the impact on costs, schedules, and national security by the failure of systems to interoperate successfully.

The increasing cost of software for military purposes has also raised an awareness of the cost of rebuilding software for similar purposes again and again. Much attention is now being focused on ways to reuse software, and as a result it is important to identify those software qualities which enhance its reusability.

1.3 SOFTWARE QUALITY FRAMEWORK

The goal of the quality metrics is to enable a software acquistion manager to specify the types and degrees of qualities desired in the end product and to predict those qualities during the development process. McCall and et. al. had established a framework for viewing software quality. Figure 1.3-1 is a simple depiction of this framework, showing a hierarchical relationship between the quality factors, and the quality criteria. The framework has now futher expanded and revised into new model. Figure 1.3-2 is a simple depiction of this new model, showing a hierarchical relationship between the quality factors, i.e. interoperability and reusability, the quality criteria, and the quality metrics. The quality factors are user-oriented terms representing concerns of the acquisition manager or product user. Quality factors are used to specify the type of quality desired. Criteria are software-oriented terms representing attributes of the software which, if present in the software, indicate the presence of a type of quality (a quality factor). Metrics are software-oriented phrases or sentences which ask questions concerning details of an attribute (criteria) of the software. Answers to the questions enable quantification of the degree of presence of quality criteria and, hence, quality factors.

Reusability of software requires that the software be understandable, flexible, modifiable, adaptable, applicable and accessible. Simplicity, system clarity, and self-descriptiveness criteria will enhance the understandability. Generality, machine and software system independences, application independence and modularity will improve the flexibility, modifibility and adaptability. Functional scope is an application dependent criteria, which will depend on what kind of functional requirement is needed in the reusable application. Well-structured documentation and no-control access will improve the accessibility. Thus the simplicity criterion was added to the reusability criteria. Software system independence and machine independence were consolidated into and replaced by the term independence. This enables consideration of modular and functional independence. Overall, of the five original criteria, three remain identical, two were consolidated into one criterion with a new name, one was added, and three are different.

Interoperability is a function of how well one embedded software system matches the system with which it is to interoperate, as well as a function of the ease with which the software can be changed or expanded to interoperate with another system. The system matching is measured by the commonality and independence, and the new criterion of

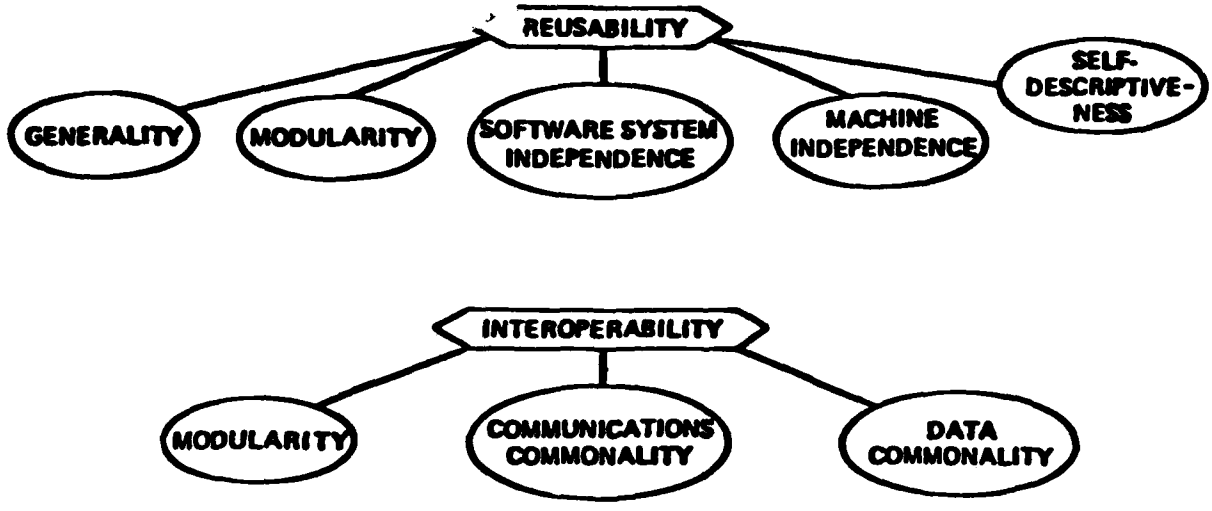


Figure 1.3-1. Software Quality Framework (Old)

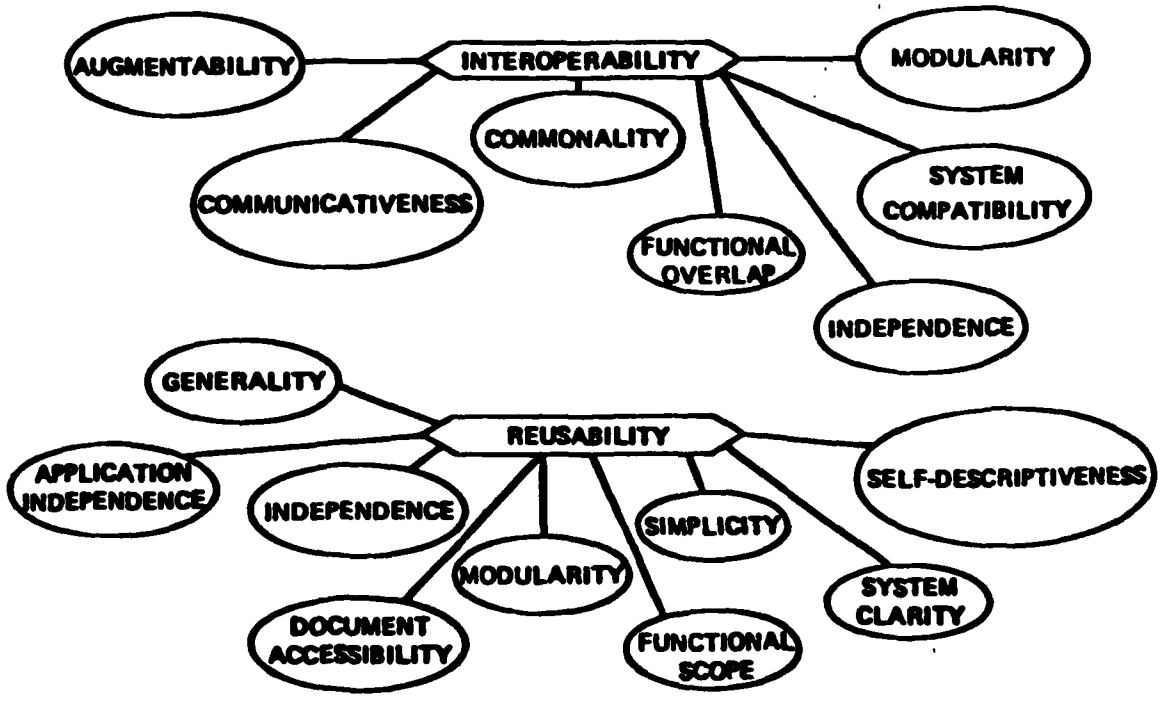


Figure 1.3-2. Software Quality Model (New)

system compatibility. The ability to change or expand the software system is measured by the original criterion of modularity, the modified criterion of augmentability, and the new criterion of functional overlap.

In summary, the new enhanced software quality measurement framework has the new features as showed in the Table 1.1-1

TABLE 1.1-1 Enhanced Software Quality Measurement Framework

FACTOR	STATUS	CRITERIA	METRICS
INTEROPERABILITY	ADD	5 (2 NEW)	13 (7 NEW)
	DELETE	1	1
	MODIFY	3	6
REUSABILITY	NET	3 → 7	4 → 16
	ADD	5 (4 NEW)	20 (16 NEW)
	DELETE	1	1
	MODIFY	1	2
	NET	5 → 9	9 → 28

1.4 TECHNICAL APPROACH

The approach to this problem was to use the previous work accomplished by RADC (see section 1.2) as well as previous Boeing software quality metrics research as a baseline. The technical approach was divided into a series of eight tasks in order to accomplish the objectives and the requirements of the statement of work. The following task flow is shown in Figure 1.4-1.

Task 1: Identify interoperability and reusability characteristics

Task 2: Develop analysis and prediction methodology

Task 3: Develop design guidelines

Task 4: Develop software quality measurement framework

Task 5: Assess impact on Automated Measurement Tool (AMT)

Task 6: Collect data

Task 7: Validate metrics

Task 8: Integrate results for Final Report and Handbook

In task 1 the distinguishing characteristics of interoperability and reusability were identified. Details of requirement specification, design, and development practices that are related to the quality factors of interoperability and reusability were identified. Tradeoffs between various quality factors were described, and concrete recommendations for implementing requirements for interoperability and reusability in the Air Force environments were developed.

In task 2 a framework for the analysis and prediction of interoperability and reusability was constructed. Since the two quality factors of interoperability and reusability are so different the metric framework for each was developed separately. Reusability could be assessed and predicted at the module as well as system level, but interoperability has

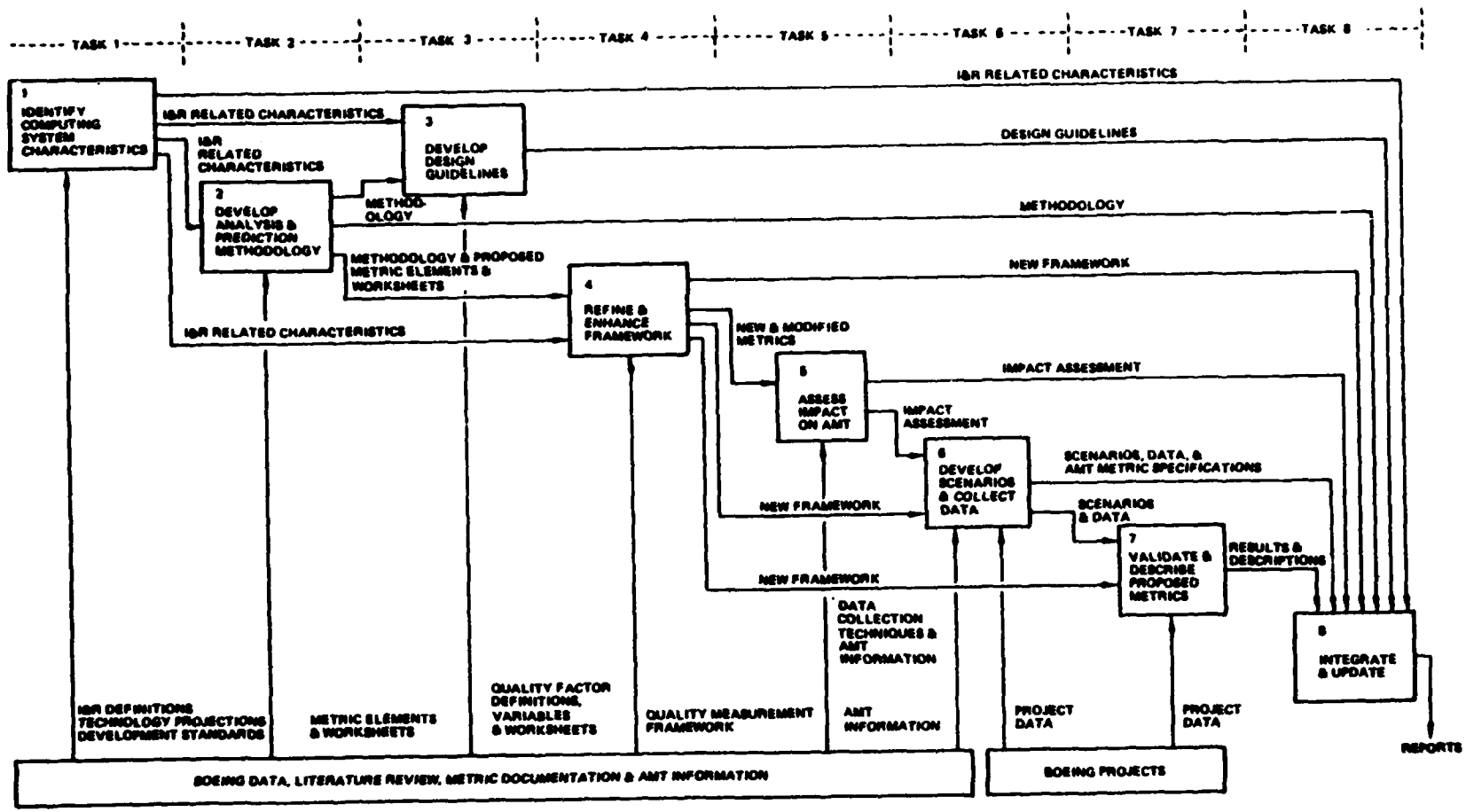


Figure 1.4-1 *Software Interoperability and Reusability Task Flow*

meaning only at the system level. These frameworks and the considerations which guide them are discussed separately in detail in sections 2.2 and 2.3.

In task 3 design guidelines for facilitating a choice among possible design implementations were considered. These guidelines for interoperability and reusability considerations appear in sections 2.2 and 2.3 respectively.

In task 4 the software quality metric framework defined in "Software Quality Metrics Enhancement" and "Software Quality Measurement Manual" (RADC-TR-80-109, Volumes I & II) were expanded and refined to enhance the quality factors of interoperability and reusability. The manual was also reformatted and reorganized to enhance readability and usability and is included as Volume II of this report (also see section 2.5 of this volume.)

In task 5 the impact of this research on the Automated Quality Measurement Tool (AMT) was assessed. The result of this assessment appears in section 2.4.

In task 6 the methodology for data collection was developed, and was implemented in the form of worksheets. (These worksheets are included in Volume II as Appendix A.) Data was collected from various software development projects; the specific software examined was determined by the influence interoperability or reusability had played on that software. As a result, different modules were examined for interoperability than for reusability, even though both might come from the same project. (See sections 2.2.5 and 2.3.8)

In task 7 the data collected in task 6 was used to validate the metrics for interoperability and reusability. The validation techniques used in RADC-TR-77-369 were reviewed and compared against other numerical analysis and statistical correlation methods in order to select the validation methodology most appropriate to the characteristics of the data. (See sections 2.2.6 and 2.3.9)

SECTION 2
TECHNICAL ACCOMPLISHMENTS

2.1 SOFTWARE INTEROPERABILITY AND REUSABILITY

The software quality framework developed in (McCall 79-1) identified eleven quality factors and established a system of criteria and metrics through which the degree of a quality factor possessed by a software system or module could be measured or predicted. The framework for the quality factors of interoperability and reusability was not fully developed by that research however.

This section establishes a perspective on these two quality factors from which a quality assessment system can be developed.

2.1.1 Top Level Framework

Figure 2.1-1 shows the top level framework within which software interoperability and reusability quality factors were investigated. From a user standpoint a software system must possess some minimum degree of the other quality factors otherwise the software would not be considered worthy of reuse. Similarly when a user considers putting two systems together, he is very concerned about the end result; will the new system reflect the best, or the worst, qualities of each subsystem. Interoperability is similar to reusability because when a new system is created from two existing systems, they are being reused in a context that differs from the one in which they were developed.

Table 2.1-1 classifies quality factors as either user oriented or production oriented. This is a departure from the McCall classification of product operation, product revision, and product transition. The product operation quality factors correspond to the user oriented factors. The product revision plus the product transition quality factors correspond to the production oriented factors. The selected classification emphasizes the differences between the orientation of what the user wants and what the software developer may do to cost effectively produce software. Of the six production oriented quality factors, interoperability and reusability are best suited to the prediction of user oriented quality factors because they capitalize on the fruits of previously developed software.

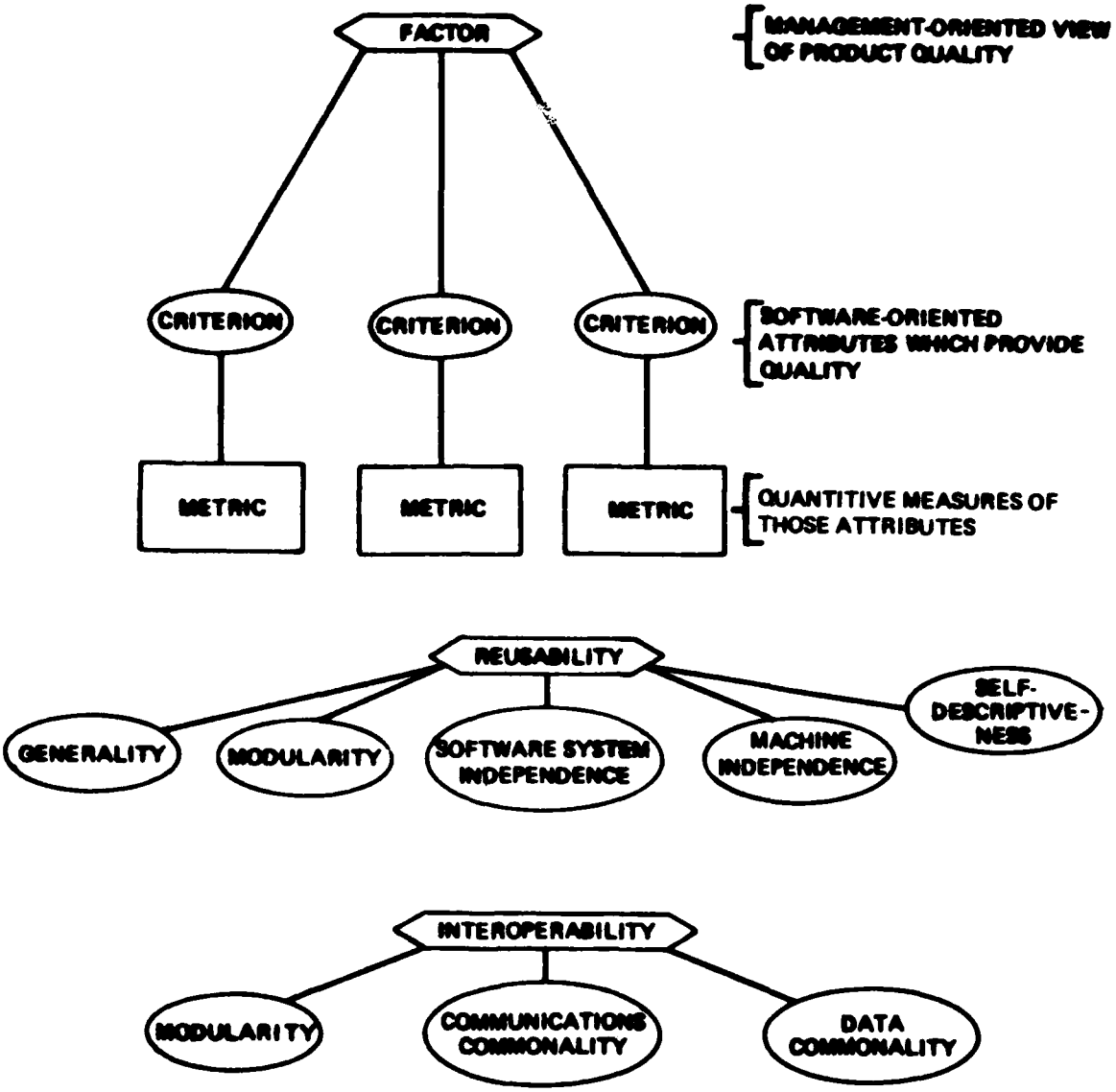


Figure 2.1-1 *Software Quality Framework (Old)*

Table 2.1-1 Classification of Quality Factors

USER ORIENTED QUALITY FACTORS	PRODUCTION ORIENTED QUALITY FACTORS
Correctness	Maintainability
Efficiency	Flexibility
Integrity	Portability
Reliability	Testability
Usability	Interoperability
	Reusability

2.1.2 Cost/Benefit Perspective

The major thrust of this report involves the determination of the interoperability and reusability potential of existing software. The reason for this is that it is assumed that interoperability and reusability quality factor ratings can be improved only at increased cost. Hence an acquisition manager would be reluctant to apply resources that would benefit some future unknown application. The developers would probably choose to improve some other quality factors since they would not necessarily benefit from improving the software interoperability and reusability.

One exception is when an organization specifies programming techniques that improve interoperability and reusability quality factors in order to reduce later development costs. However, the added costs of improving interoperability and reusability may not be justified if the software is not subsequently reused or coupled with another system. Since this cannot usually be determined in advance, any relatively large cost penalties for improving interoperability and reusability will probably be avoided.

Another aspect is when software is designed to be interoperable or reusable, the conditions of the usage will not be known in advance. Therefore, all factors which affect the software's potential to be reused or to interoperate must be addressed, which may be quite costly. However, if existing software is being assessed for reuse or interoperation with other systems, the software will be known to meet the needs of the new application. Also the environment will be known, allowing the identification of unimportant interoperability and reusability quality factors. For example, if the software will be reused on the same computer, word size factors are unimportant.

For software to be reused, or coupled with other software, the cost of its use must be less than the cost of new software development. The largest potential benefit of software interoperability and reusability quality metrics is in estimating those costs and reducing the risk in a decision to utilize existing software in a new system.

2.1.3 Quality Concerns

The user has two primary concerns relative to reusability and interoperability. They are

1) What is the cost of reusing software or making software systems interoperable?

2) What is the operating cost of the resulting software system?

The first question refers to development costs. Where the quality factors portability, flexibility, and testability are the most important. During operation, efficiency, maintainability, and reliability have the greatest cost impact.

Usability, correctness, and integrity are usually of greatest importance during the initial committment to reuse the software or make it interoperable with another system.

Figure 2.1-2 shows how the factors of interoperability and reusability may be used to estimate software development costs. The new application will have both functional and quality requirements. The functional requirements are used to set criteria for reusability and interoperability. The existing data base of software is assessed and a subset is selected and evaluated for its conformance to user quality requirements and its potential of reuse or interoperation. Quality upgrade needs are assessed by comparison of the required software quality profile with the existing software quality profile. The quality upgrade needs may be converted to quality upgrade costs through use of curves that relate quality enhancement to cost.

The interoperability and reusability ratings provide estimates of the amount of change needed to make existing software functionally acceptable. New functions are identified by comparison of existing functions with the application requirements. The combined assessment yields the software quantity needs. The change quantity needs may be converted to quantity upgrade costs through the use of curves that relate to productivity. The quality upgrade costs and the quantity upgrade costs are then used to predict new product software costs.

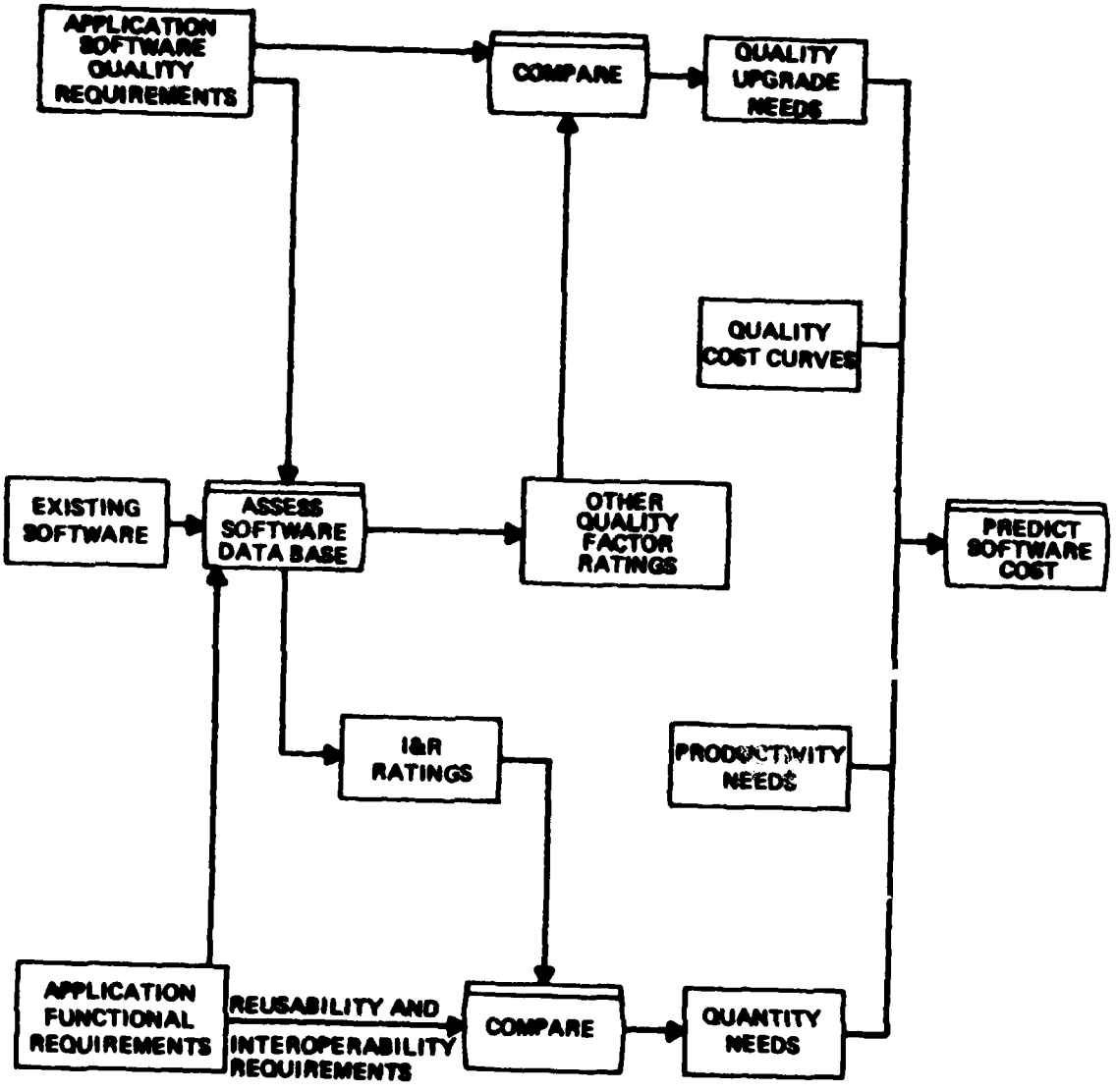


Figure 2.1-2 *Operational Use of the Interoperability and Reusability Quality Factors*

2.2 INTEROPERABILITY

The software quality factor "interoperability" is defined in the Software Quality Measurement Manual (McCall 79-1) as "the effort required to couple one system with another." Coupling includes linking two programs to interoperate on a single computer or linking programs on separate computers to interoperate. The quality factor interoperability is therefore important when:

1) retrofitting two or more previously developed systems.
2) developing new systems independently that will interoperate with each other.
3) developing a system with the expectation that it will eventually interoperate with an, as yet, undefined future system.

When compared to the other quality factors, interoperability shows several unique aspects that are not considered in the other factors. Each of the other factors is defined in terms of "a (computer) program" or "(computer) software." In contrast, interoperability is defined in terms of "a system." There is an immediate implication that interoperability considers something more global than the other factors, for "system" implies something more than "software" – it implies the framework of hardware, communication links, and human interaction in which the software is embedded. Thus, the approach to define and validate metrics for systems was different in kind from the approaches used strictly for software.

Interoperability was approached by first reviewing definitions of interoperability (2.2.2.4), then by building a framework of criteria and metrics (2.2.2.5). Because interoperability includes a large range of problems beyond any one researcher's experience, various managers and engineers whose experience included work on projects with interoperability considerations were interviewed. They were asked to talk about interoperability problems they had encountered, and then were asked to review the proposed framework for interoperability metrics. As a result of the interviews, significant changes were made in both the framework and our data worksheets. The interviews provided so much additional insight into the range of interoperability problems that it prompted an entire section (2.2.2) on the results.

Section 2.2.3 is a description of the metric framework for interoperability and its theoretical basis. It also includes a discussion of the reasons various metrics were dropped and others added. Section 2.2.4 discusses tradeoffs between interoperability and other quality factors. This section is followed by a description of the data collection methodology (2.2.5). The next two sections present the validation results (2.2.6), the conclusions and recommendations (2.2.7).

2.2.1 Key Concepts

The two key interoperability concepts are sharing of data and sharing of functions between two or more systems. These concepts are referred to as data coupling and functional overlap, respectively.

2.2.1.1 Data Coupling

Systems may be data coupled through sharing a common data structure, data management system, communication link or network. The two extremes of data coupling are "loosely coupled" and "tightly coupled".

Two software systems are loosely coupled when specific data items are shared through use of a communication framework that passes the items between the systems. Generally the amount of data shared is relatively small and the occurrence of data sharing is infrequent. Issues arising in such systems include increased communications overhead and delays, the impact of shared resources on system performance, the impact of adapting the two systems to a unified executive, the potential for software redundancy (duplicated code) and/or conflicting results between the two systems, and problems in timing and synchronization.

Two software systems are tightly coupled when they share a common data structure or database and a mechanism is required to prevent conflicts between the systems during data access and updates. Generally, the amount of data shared is relatively large and the occurrence of data sharing is frequent. Issues arising in such systems are similar to those of loosely coupled systems, but to different degrees. Communication overhead and delays may be less of a problem, but the issues of conflicting results, timing and synchronization may be significantly more important.

2.2.1.2 Functional Overlap

Systems may be coupled through the sharing of common functions, referred to as functional overlap. If two systems have few functions in common the degree of functional overlap is low. If they have a large number of functions in common the degree of functional overlap is high. If the two systems have in common some similar, but not identical functions, problems may arise from conflicting results. Other issues arising in such systems are increased maintenance requirements, additional complexity, timing and synchronization and the possible need for an external control mode.

2.2.2 System Characteristics

This section identifies the characteristics which impact a systems ability to interoperate. Table 2.2-1 details all of the system characteristics that were considered. Those marked "accepted" are described with respect to software interoperability. Those marked "rejected" were considered to be redundant or of lesser importance with respect to software interoperability.

Table 2.2-1 Characteristics that Impact Software Interoperability

	Accept	Reject
Accessibility		x
Algorithms	x	
Precision of algorithms		
Computer Architecture		
Memory and mass storage	x	
Microcode		x
Standard Features		x
Word size	x	
Data Management Methodologies	x	
Access Methods		
Database Management Systems		
Mass storage devices		
Data Structures	x	
Element association		
Structure types		
Documentation (of second system)		x
Domain Generality		x
Environment Dependence		
Data Dependence		x
Machine Dependence		x
Software System Dependence		x
External Control Modes	x	
Interactive		
Batch		
Computer		
Combined Control Language		
Fault tolerance	x	
Fault containment		
Fault detection		
Fault diagnosis		
Fault recovery		

Table 2.2-1 (Cont.)

	Accept	Reject
Functional Overlap	x	
Hardware		x
Analog		
Digital		
Human Engineering	x	
I/O Protocol	x	
Interface message processor		
Host to host protocol		
Data transfer protocol		
File transfer protocol		
Graphics protocol		
RJE protocol		
Languages	x	
Modularity	x	
Interface Complexity		
Coupling		
Output Mode	x	
Scope of functions		x
Security	x	
System Architecture		x
System Availability	x	
System Size	x	
Timing	x	

2.2.2.1 Definition or Meaning of Accepted Characteristics

Algorithms	- A series of operations to achieve the desired result.
Computer Architecture	
Memory and Mass Storage	- The configuration of main memory and secondary storage. Virtual memory is an issue.
Word Size	- The number of bits in a word, usually stored and handled as a unit.
Data Management Methodologies	- The methods used to manage, store and manipulate data.
Data Structure	- A structure to allow organized access and storage of internal data (e.g. lists, trees).
External Control Modes	- The means of providing user control over a system.
Fault Tolerance	- The ability to detect, contain, diagnose and recover from faults.
Functional Overlap	- A comparison between two systems determines the number of functions common to both systems.
Human Engineering	- The measure of how demanding it is to use the system.
I/O Protocol	- Specifications that describe I/O standards and procedures.
Languages	- Systems of symbols which can be used to provide instructions to a computer such as the computer input set, assembler and HOL languages.

Modularity	- Those attributes of the software that provide a structure of highly independent modules.
Output Mode	- The type of output, to mass storage devices or other peripherals, printers, terminals or graphics.
Security	- Protection of classified computer system components, software and data.
System Availability	- The percentage of time a system is free from component failures.
System Size	- The total amount of system resources.
Timing	- The timing requirements for a system operation.

2.2.2.2 Impact of Characteristics on Software Interoperability

Characteristics of the individual systems have an impact on the effort required to make them work together. The following summarizes the effect that differences in these characteristics between two systems, may have on the cost of making them interoperable. The two key characteristics functional overlap and coupling will be discussed in section 2.2.2.3 in relation to the other characteristics.

2.2.2.2.1 Algorithms

An algorithm is a plan made up of a series of operations that, when programmed and executed, will produce the desired data or alterations to data or data structure. If two systems share the same function, variations in algorithms, age of the data or precision of the machine may cause unacceptable differences in the results.

2.2.2.2.2 Computer Architecture

Memory and Mass Storage

Many aspects of memory and mass storage architecture are covered under other system characteristics discussions i.e., Software System Dependence, Machine Dependence of Code and System Size. Special problems of memory and mass storage are virtual memory versus real memory and that of access to mass storage.

Virtual Storage machines are essentially unlimited core machines. Very large programs can be handled via paging facilities. Problems occur when using conventional programming techniques to develop programs for use on a Virtual Storage machine.

Overuse of paging facilities result in very slow programs. Programs most appropriate for rehosting on a Virtual Storage machine are modular and able to execute in a limited portion of the machine. Program functions should execute in a manner that once core is set up, all the processing is completed before the configuration changes.

Mass storage access is also a problem. If a system utilizes an access method that is not supported by the host system, radical redesign may be required to make the systems work together.

Word Size

This is one of the physical characteristics of systems that must be considered when studying system interoperability. Each computer utilizes an operating system which is the linkage between the application software and the hardware. Computer system designers try to optimize the system to keep operating costs at a minimum. The result has been that many systems have different word sizes. Word sizes affect the accuracy of computations and the addressability of the memory locations. These differences impact the interoperability of a system with respect to the system it is intended to be merged with.

Computer accuracy in floating point calculations is dependent on the length of the word. Longer words will carry more significant figures and be less affected by round off errors.

Software with many computational algorithms designed to run on a 60 bit word sized machine will not transport to a 16 bit word sized minicomputer without extensive work. On the other hand it may be quite simple to transport in the other direction. A problem could occur from a space point of view if a great quantity of variables were used on the 16 bit word machine in conjunction with a larger program.

Addressability is the other area of potential problems. A 16 bit word inherently provides core addressing to the 16 bit level. Some machines only provide addressing to the byte level which can be six or eight bits. Others address on the half word. Obviously a program that was designed to take advantage of the addressability of the 16 bit word computer would require extensive rework to be usable on the larger word size machine.

2.2.2.2.3 Data Management Methodologies

Data management is the combination of software with data structures to allow efficient manipulation of data. A variety of data base management systems exist employing different data structures, access methods and mass storage devices. Access methods used include direct techniques like attribute or content addressing, indirect techniques like ordered positioning, chains or directories, or search techniques like sequential, binary or calculated searches. Two systems which interoperate through data management require common data structures, common access methods and compatible mass storage devices. A mechanism may be required to prevent conflicts during data access and updates.

2.2.2.2.4 Data Structures

Data structures are used to organize data for easier manipulation. Within a data structure elements may be associated with each other by means of labels, locations, keys, pointers or links. Typical structures might include tables, linked lists, multiple lists, rings or trees. For two systems to interoperate they may require that element associations be altered and that a more complex combined data structure be devised. Response times for both systems could be slowed by the larger amounts of data in a combined structure.

2.2.2.2.5 External Control Modes

The external control over a system may be interactive, batch, or by another system. The coupling of two interactive systems may result in the need for a common or combined control language. The coupling of batch systems may require the use of mutually compatible mass storage devices. The section on Output Mode discusses external control by other computer systems.

2.2.2.2.6 Fault Tolerance

A fault tolerant system will incorporate elements of fault detection, fault containment, fault diagnosis and fault recovery. The amount of fault tolerance incorporated into a system depends on a tradeoff between the future cost associated with failures versus the present cost of adding the tolerance.

When coupling two systems, one or both may require the addition of fault tolerant features. An advantage to be gained by interoperable systems which reside on different computers is that should one computer fail, the other system may be able to save some of its data, take over portions of its calculations, perform some diagnosis and aid in system recovery. A disadvantage is that the failure of one computer may render the other computer system inoperable.

2.2.2.2.7 Functional Overlap

Refer to section 2.2.2.3.

2.2.2.2.8 Human Engineering

Human engineering is a characteristic associated with the ease to which users can interface with a system. The specific area of interest is how demanding the system is to use. A system that can be easily operated by users with little experience or training is better human engineered than one that requires a high degree of training.

2.2.2.2.9 I/O Protocols

These are specifications that describe the information necessary to connect to a host computer system such as in ARPANET. Computers typically differ from one another in

type, speed, word length, operating systems, etc. Connecting host to host usually requires an interface message processor (IMP) for each computer. The IMP's are then connected. The Host - IMP interface is the first level protocol. This protocol is not sufficient to specify meaningful communication between processes running on dissimilar hosts. The next level specifies methods of establishing communication paths, managing buffer space and providing a method of interrupts. This is known as Host-to-Host protocol. Further layers of protocol include initial connection which provides a convenient standard for processes to gain simultaneous access to some specific process. A telecommunications network protocol provides mapping of terminals into a network to facilitate communication between a terminal at one host site and a terminal serving process at another. A data transfer protocol specifies standard methods of formatting data for shipment through a network. File transfer protocols specify methods of reading and writing and updating files stored at a remote host.

Graphics protocols specify a means of exchanging graphics display information. Remote job service protocol specifies methods for submitting input and obtaining output from, and exercising control over, hosts which provide batch processing service.

Each of the above protocol specifications provide essential information required for connecting a system. If these are available the merging of additional components is straight forward. If they are not available it is probable that they will have to be prepared prior to any extensive system modification and reviewed thoroughly in determining the effort required to couple the systems. If the systems use different protocols one or both must be changed.

2.2.2.2.10 Languages

Programming languages in use today can generally be divided into three categories, machine oriented (i.e. assembler), procedure oriented (i.e. Fortran, Cobol, Jovial) and problem oriented (i.e. Mark IV, RPG).

Machine oriented languages strongly reflect the features of the particular computer, while higher level languages are more standardized. Functional coupling could be difficult if one or both of the systems were programmed in a machine oriented language. Languages may also affect systems which require data coupling even if using a higher level language. For example, on the CDC Cyber 70, a Fortran program will read

imbedded blanks in numeric hollerith data as zeros while embedded blanks cause an error for Cobol programs, furthermore the Fortran program requires a decimal place on floating point numbers, but Cobol can assume a decimal place. Additional problems may occur between other languages.

2.2.2.2.11 Modularity

Refer to section 2.2.2.3.

2.2.2.2.12 Output Mode

The important aspects of this characteristic are output mode, output content and output format.

Output may be to a mass storage device i.e., drum, disk, tape. It may go directly to peripherals for processing i.e., printing, card punching, tape punching. Or, it may be sent to a users terminal and displayed graphically. Collecting and analyzing this kind of data for each system supports determination of how much effort will be required to convert the output data of the systems to couple them.

Formats of the output may have to be revised to be consistent in the final system. Reading computer outputs is simplified if the layouts of different reports are made similar.

Outputs from two systems could be related or duplicated. It is generally necessary to review output reports and integrate data from both systems into a single set of output reports that provide maximum information in a concise, clear manner.

2.2.2.2.13 Security

A special case of processing to be considered is handling of classified input and output material. Almost every instance would require the new system be classified to the highest level of the separate system. The effort of merging would not be affected as much as the handling of the program and its outputs thereafter.

2.2.2.2.14 System Availability

This is a measure of the percentage of time that a system is ready to be used or being used. This measure depends on the amount of time that a system is down for problem correction (unplanned maintenance) and for adding enhancements (planned maintenance). Availability then is clearly a function of the failure rate (unplanned), how long and how difficult it is to fix (maintainability) and planned enhancements. The combination of high failure rate and a large number of enhancements will reduce system availability significantly.

When merging two systems it is important that availability of each be known and the requirements of the new system. It is also important to understand how easy or difficult it is to modify the software. Finally the amount of planned machine maintenance must be considered. This total spectrum is required in order to predict whether the final system will meet availability requirements.

2.2.2.2.15 System Size

A system may already be utilizing a large percentage of available resources. In coupling systems, interface penalties in buffer space, high speed memory or mass storage may occur due to alterations in one or both systems algorithms, data structures or data management techniques. The alterations may require memory overlays or may be beyond the current systems capability.

2.2.2.2.16 Timing

System timing can be commonly divided into three categories, real-time, interactive and delayed or batch. A real-time system interacts with a physical environment in order to service or control that environment, which makes timing critical. In interactive systems responses are in real time but timing is not critical while in batch systems processing occurs as convenient.

In coupling systems, penalties in execution and response time may occur due to alterations in algorithms, overlays, data structures, data management techniques, or other overhead incurred by working together. These penalties may alter critical timing relationships in real-time systems or cause unacceptable delays in response times of interactive systems.

2.2.2.3 Data Coupling and Functional Overlap vs. System Characteristics

This section examines the relationships that exist between the key concepts of software interoperability (data coupling and functional overlap) and other system characteristics. Table 2.2-2 presents selected system characteristics on one axis and key concepts on the other. Areas where relevant relationships exist are marked with an item number to reference following discussions.

Table 2.2-2 Interoperability Concepts vs. System Characteristics

System Characteristics	Interoperability Key Concepts			
	Tightly Coupled	Loosely Coupled	Large Functional Overlap	Small Functional Overlap
Algorithms			1	
Computer Architecture				
Memory and Mass Storage	2			
Word Size				
Data Management Methodologies	3			
Data Structure	4			
External Control Modes			5	
Fault Tolerance		6		
Functional Overlap				
Human Engineering				
I/O Protocol		7		
Languages			8	8
Modularity				
Output Mode				
Security	9	9	9	9
System Availability				
System Size				
Timing	10	10	10	10

1. The same function in two independent systems may use different algorithms. One algorithm may be designed for speed and the other algorithm for accuracy. If the requirements are the same for both functions and the code is modularized (high functional strength), it may be possible to share one of these functions. However, if each system module contains many functions (low functional strength) it may be difficult to share either function. It may be necessary to rewrite the function involved to achieve coupling.

Figure 2.2-1 summarizes the relationship between functional overlap, module strength and the cost of coupling two systems. The classification scheme for module strength is that defined by (Meyers 75). The cost for informational and functional is linear because these types of modules have high independence from other modules, are easy to use in other programs, and are easy to extend. The others are non-linear because they lack these properties to various degrees.

The test for modularity is presented in decision table format in (Meyers 75). This table is shown here as Table 2.2-3. The decision table provides a simple means of determining the strength of a module. The criteria are primarily judgmental, but are amenable to interactive analysis.

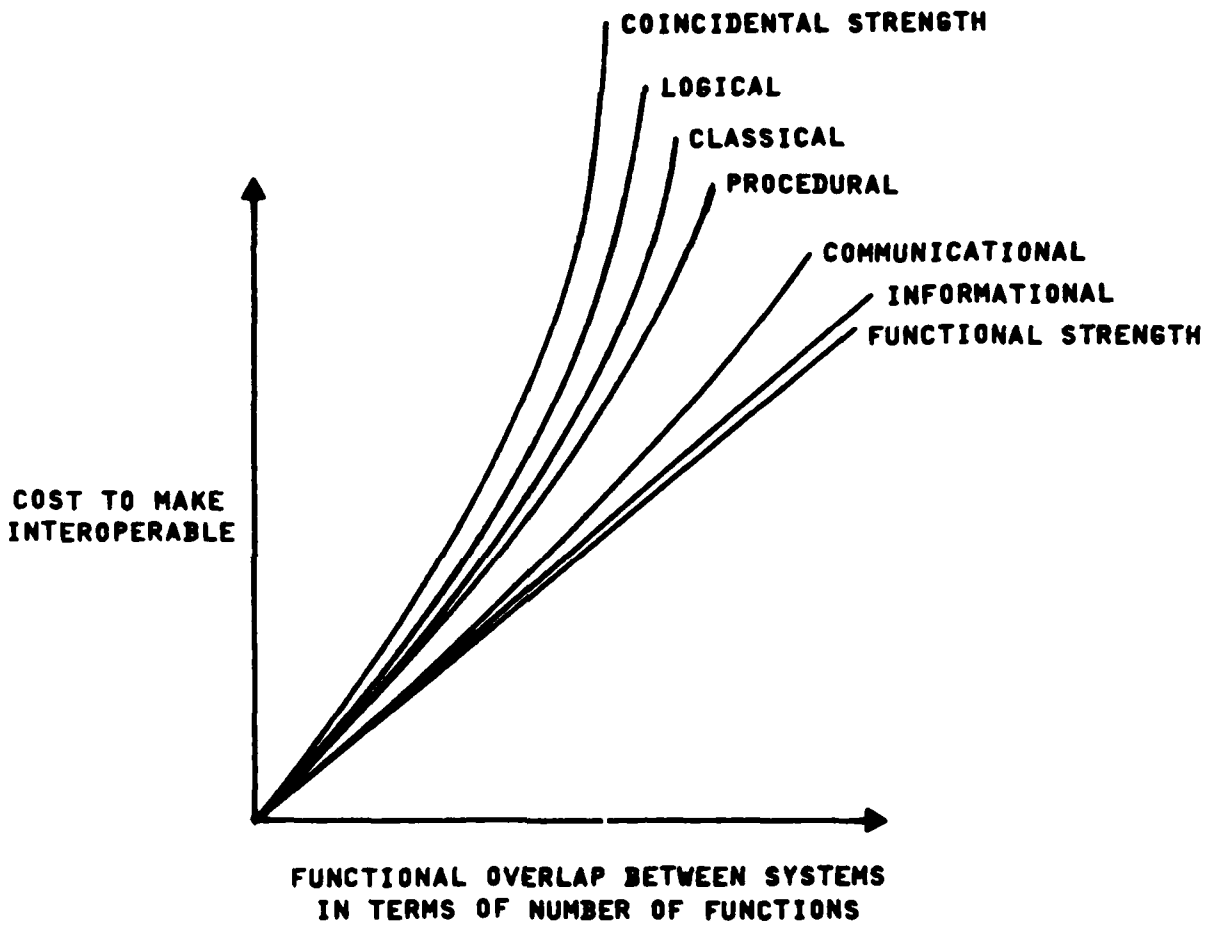


Figure 2.2-1 Cost to Make Interoperable Depends on Functional Overlap and Module Strength

CRITERIA	DIFFICULT TO DESCRIBE THE MODULE'S FUNCTION(S)	Y	N	N	N	N	N	N	N
	MODULE PERFORMS MORE THAN ONE FUNCTION			Y	Y	Y	Y	Y	N
	ONLY ONE FUNCTION PERFORMED PER INVOCATION			Y	N	N	N	Y	
	EACH FUNCTION HAS AN ENTRY POINT			N				Y	
	MODULE PERFORMS RELATED CLASS OF FUNCTIONS		N	Y	Y				
	FUNCTIONS ARE RELATED TO PROBLEM'S PROCEDURE				N	Y	Y		
	ALL OF THE FUNCTIONS USE THE SAME DATA					N	Y	Y	
FUNCTIONAL STRENGTH	COINCIDENTAL	X	X						
	LOGICAL			X					
	CLASSICAL				X				
	PROCEDURAL					X			
	COMMUNICATIONAL						X		
	INFORMATIONAL							X	
	FUNCTIONAL								X

Table 2.2-3 Seven Criteria Determine Functional Strength

2. The memory required for shared resources and a unified executive may exceed machine capacity. Paging and swapping facilities may be too slow for a combined systems.

3. To tightly couple two systems to share a common DBMS requires compatibility of their data bases. A common sort key or common element association is necessary. Record processing codes need to agree.

4. To achieve a tightly coupled system through a shared data structure, it may be necessary to create previously unneeded portions of the data structure for use by one of the systems. If multiple calling modules are operating it may be necessary to save and restore portions of the data structure. Program modules may have access to unnecessary data which can cause inadvertant side effects or compromise data integrity.

5. Two systems with large functional overlap require a combined external control mode.

6. Fault tolerance can be added to loosely coupled systems at low cost.

7. Loosely coupled systems require commonality of I/O protocol for the interface message processor, telecommunications network and data transfer.

8. Sharing of functions between systems which utilize different assembly languages is difficult due to the machine dependence of the languages.

9. Coupling brings a new dimension to the problem of security, as it adds complexity due to the number of systems and levels of security within each system. This may cause a severe impact in terms of cost, design and schedules.

10. In coupling systems, sacrifices in execution and response time may occur due to alterations or overhead incurred in coupling the systems.

2.2.2.4 Interoperability Definitions

As noted above, the working definition of interoperability is "the effort required to couple one system with another." This definition was compared to those cited in technical papers and reports on "interoperability" listed below

1.) The ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces and to use the services so exchanged to enable them to operate effectively together.

DODD 2010.6 Standardization and Interoperability of Weapon Systems and Equipment Within the North Atlantic Treaty Organization (NATO), 11 March 1977.

2.) (The ability of one service's system to receive and process intelligible information of mutual interest transmitted by another service's system.

JINTACCS INTEROPERABILITY
ref-PM99 21 DEC 1974 HQDA

3.) The ability of one system to receive and process intelligible information of mutual interest transmitted by another system.

INTEROPERABILITY VIA EMULATION
Ingrid A. Eldridge
Proceedings of the 1978 Summer Computer Simulation Conference
July 24-26, 1978
Los Angeles, Calif.

Interoperability is conceived as a good quality, in the same way we speak of reliability. These definitions correspond to common sense notions. Defining "interoperability" in terms of effort is an indirect way to look at this quality factor.

There are further problems with the McCall definition. The words "system" and "couple" are never clearly defined. This definition, taken literally, includes only the connectivity

and compatibility issues of interoperability; it implies only equipment level considerations. Two persons using exactly the same UHF transmitters on the same frequency may not be able to interoperate, particularly if one person speaks only Chinese, the other speaks only Arabic. Hardware compatibility does not assure interoperability. Interoperability is achieved when both persons can transmit and receive information of mutual use and understandability.

The NATO definition is also unsatisfactory because it stresses standardization, especially of equipment. The emphasis on standardization of hardware and software overlooks the content of the messages and the differing operational requirements that affect interoperability.

The JINTACCS and DoDD 2010.6 definitions are preferred. This preference is based not only on review of technical publications, but also on interviews with many managers and engineers who have faced interoperability problems. These definitions seem to most accurately define the ultimate meaning of interoperability.

The working definition was chosen by the original researchers (McCall, et. al.) because it gave them something they could measure; most projects have some estimate of manpower expended. This, in turn, provided a measure against which the metrics could be correlated. This approach was attractive, because it would provide a method to predict cost.

2.2.2.5 Interoperability Interviews

The wide range of interpretations of interoperability in the literature suggested that a comparably wide range of interoperability experience must exist. So managers and software engineers who had had experience with interoperability requirements or problems were interviewed. Twenty interviews were held. As a result, the concept of interoperability and of which criteria most influence interoperability were considerably altered.

The interviewees had an enormous range of experience. Projects varied from microprocessor systems to very large embedded software systems containing over a million lines of high order source code. The interoperability problems also varied from those within

small sub-systems to those between several large command, control and communication (C3) systems of different nations. As a result of the interviews the common essence of these problems was extracted and stated in general terms. The following guidelines represent a distillation of all the recommendations received on how best to assure interoperability.

1. The most critical decisions with regard to interoperability are made during the specification of the system level requirements and interface requirements.

2. The specification of operational procedures and interface definitions should be as explicit as possible, rather than general, to assure interoperability. This point was stressed by most of the project personnel interviewed:

> It is not how generally the interface requirements are specified, but rather how specifically. The more precisely the interface is defined, in terms of its protocols, message format, and message content, the more likely interoperability will be achieved. Thus metrics should measure the *precision and completeness of communication* requirements, and their adherence to standards.

3. The definition and understanding of the operational procedures for using the system is as important as the technical interface definition.

4. Hardware interoperability, while important, is only the first step in achieving system interoperability; and it is a comparatively minor problem.

5. 'Off-line interoperability,' such as the requirement that two systems must use the same data reduction and analysis software is also important. A ground based data reduction and analysis system may have requirements to work with several surveillance systems, so their data can be processed, reduced, analyzed, and compared. Neither surveillance system in this example, has the requirement to interface with each other. But both systems have the requirements to prepare mission recording tapes that can be processed by the ground based data reduction system. This is off-line interoperability. This distinction, however, appears not to be important in metric calculations, for

the basic interface considerations are almost identical. Transmission rate considerations become a matter of tape motion speed; communication and data considerations become the relatively simple factors of number of tracks, formatting, blocking, and data representation standards. Thus, a separate set of criteria and factors were not developed for the off-line situation.

6. The availability of accurate and up-to-date documentation is very important.

7. Timing requirements are as important as message format and content standards. There are two major timing categories: first, the response time of one system to a message from another interoperating system; second, the 'data stale' requirements. An example of the latter would be a system requirement that data older than 20 seconds is of no use because it is 'stale', or too old to be of value. Such a data stale requirement may dictate the interrupt structure of an executive, or impose limits on the buffering of data.

8. A common vocabulary between users of both systems is of major importance.

9. A simple interface is important, but an explicitly defined interface is crucial.

10. The human interface should be as good as possible, since the human is sometimes the only interface between systems.

Interoperability requirements are sometimes more subtle than they first appear. For example, E-3A was designed to replace BUIC, which in turn was designed to replace SAGE. Both older systems had interface definitions that were, to some extent, technically obsolete. Yet, since the older systems were also, to some extent, still operational, E-3A was required to interoperate with both of them; which further perpetuated the clumsy interfaces. On the other hand, E-3A was required to interface with modern systems planned to be operational during the E-3A life span. Conflicting requirements force compromises in the efficiency of present and future interfaces. An optimal resolution of these conflicting interoperability requirements is difficult to achieve, and inevitably involves unattractive compromises.

2.2.3 Interoperability Criteria

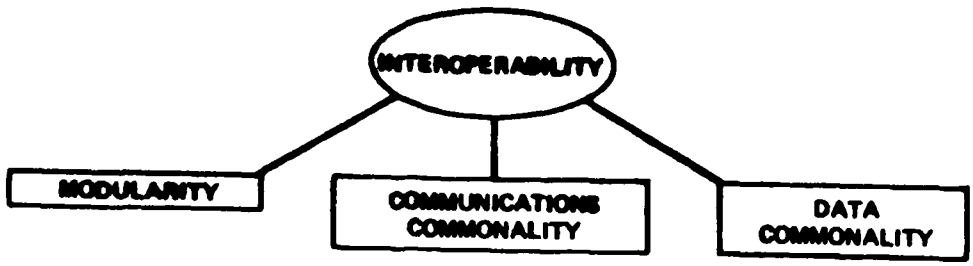
The original and the new interoperability frameworks are illustrated in Figure 2.2-2. The new framework was derived by re-examination of all criteria which could conceivably affect interoperability. This re-examination included several approaches in the consideration of the effect on interoperability by each candidate criteria and its constituent metrics.

If interoperability is considered in very basic terms, three approaches suggest themselves. The first approach is to measure the "goodnesses" within the software of a single system that would enhance interoperability. The second approach is to measure how easily the software could be changed, for some change is usually required to make the system interoperable. The third approach is to measure how well two specific systems would interoperate. Each of the approaches is discussed separately in section 2.2.3.1 thru 2.2.3.3.

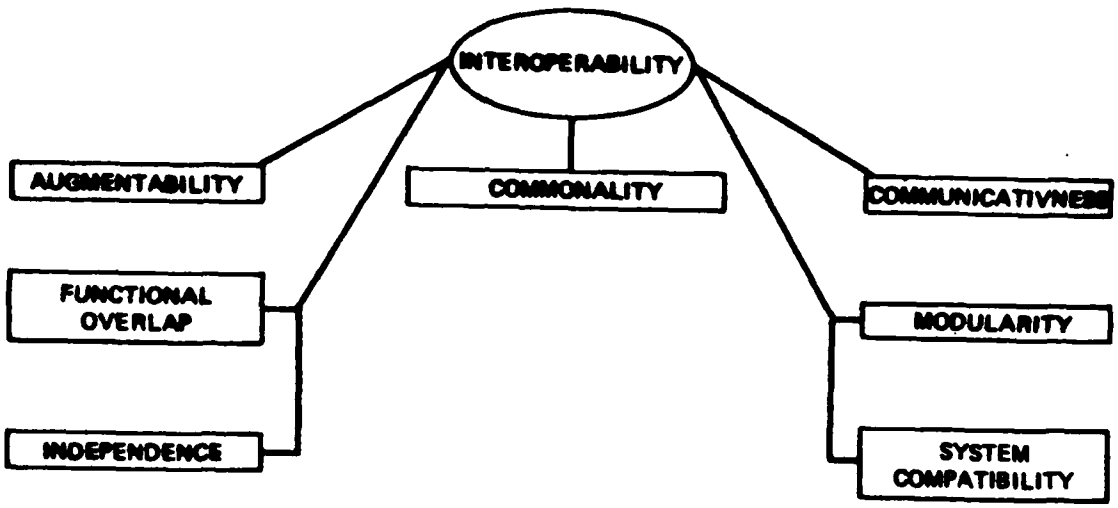
The criteria chosen for the new framework (Figure 2.2-2) are defined below.

**AUGMENTABILITY (AG)	Those attributes of the software that provide for expansion of data storage requirements or computational functions.
**COMMONALITY (CL)	Those attributes of the software that provide the use of standard protocols and interface routines.
COMMUNICATIVENESS (CM)	Those attributes of the software that provide useful inputs and outputs which can be assimilated.
*FUNCTIONAL OVERLAP (FO)	A comparison between two systems to determine the number of functions common to both systems.
**INDEPENDENCE (ID)	Attributes of the software that determine software dependency on the software environment such as software system, data system, machine, algorithm and computer architecture.

* = New
** = Revised



OLD INTEROPERABILITY FRAMEWORK



NEW INTEROPERABILITY FRAMEWORK

Figure 2.2-2 Original & New Interoperability Framework

MODULARITY (MO)	Those attributes of the software that provide a structure of highly independent modules.
*SYSTEM COMPATIBILITY (SY)	A measure of the hardware, software and communication compatibility of two systems.

* = New
** = Revised

2.2.3.1 Metrics Measuring the Interoperability of a Single System

This discussion of attributes which directly affects the interoperability of a single system has an underlying assumption. The assumption is that there are attributes that always contribute toward interoperability in the same way. That is, the "good" attributes always have a positive effect on interoperability, and the "bad" attributes always have a negative effect. Another way of stating this assumption is that there are certain fundamental relationships that govern interoperability, and that these relationships are universal and invariant. This assumption is necessary when trying to identify attributes that affect interoperability, for there would be chaos if the effect of all attributes were unpredictable.

The following criteria and metrics were proposed as having a direct and predictable effect on interoperability; each is discussed in turn. Metric codes in parentheses identify the specific metrics within each criteria. With the exception of new metrics, these identifiers refer to metrics defined in McCall's report. As a result of the interviews, several of the proposed metrics were rejected; the rejection rationale is also discussed.

COMMUNICATION COMMONALITY - The Communications Commonality checklist (CL.1) assesses the system software for 1) a definitive statement of requirements for communication with other systems (requirements phase), 2) communications protocol standards for communication with other systems, and 3) single module interface for input from another system, as well as single module interface for output to another system. These elements all contribute to the probability that the system will be interoperable, and that it will easily modified if necessary. (CL.1)

DATA COMMONALITY - The Data Commonality checklist (CL.2) assesses 1) A definitive statement for data standard representation for communication to other systems (requirements phase) and 2) translation standards among representations established and followed (design and implementation phase), as well as 3) the use of a single module for each translation. (CL.2)

MACHINE INDEPENDENCE - When the software source code is in a programming language available on other machines, is independent of character and word size of the particular machine, is relatively free from I/O references bound to the particular machine, and uses a data representation that is machine independent; then it is more likely to be interoperable with other systems. (ID.2)

SOFTWARE SYSTEM INDEPENDENCE - When the software is independent of specific operating system software or utility software, it is more likely to be interoperable with another system. (ID.1)

ANOMALY MANAGEMENT - The first two anomaly management metrics (AM.1 and AM.2) assess the error handling architecture and the ability to recover from improper input data. With these characteristics, the software system is far more likely to work with another system, since it can gracefully handle and recover from anomalies that may occur when interoperating with another system. DELETED

This criteria was deleted after interviews with personnel whose project experience included solving interoperability problems. The consensus was that error correction is normally addressed in the design of the communication link; and that the protocols are structured to compensate for the link architecture. For example, critical data may be transmitted several times to assure that it is received correctly. On the other hand, data of relatively low criticality may not be worth the effort to recover; by the time that data is reconstructed or recovered, it is stale (out of date).

Those interviewed did not agree on the subject of error tolerance. One manager thought that software should not be rated lower if it is not error tolerant. Intolerant software is usually very specific in what it will accept.

This intolerance results in a precise interface definition. An error tolerant software module that is to be made interoperable may present a greater problem, because its error tolerant design must be understood completely in order to define the interface. The interface becomes much more complex because the error tolerance adds an element of ambiguity to the interface definition. This opinion evoked strong responses, both for and against, from the other interviewees.

I/O USER INTERFACE - The human engineering design of the user interface with the system is important with respect to interoperability. The user interface with the system must be simple, intuitively consistent, and as conventional as possible. The user may serve as an important link in the interoperability chain, so his interfaces with other systems should be identical. (CM.1 User Input Interface Measure, CM.2 User Output Interface Measure)

2.2.3.2 Metrics Measuring the Adaptability of a System

These criteria and metrics were selected on the assumption that software changes will be required in order to make that system interoperable. Thus, these criteria are associated with the ease of software modification. Each criteria is discussed below.

EXPANDABILITY CRITERIA - The data storage and processor timing reserves available for expansion will affect the difficulty of making the modifications that may be required for interoperability. If timing and sizing margins are tight, it is difficult to add new capabilities within these constraints. (AG.1) DATA STORAGE EXPANSION

MODULAR IMPLEMENTATION - Software that is constructed in a modular fashion will tend to limit the impact of changes necessary for interoperability, thus making the modifications easier to assess and implement. (MO.2)

Total modularity is not required for interoperability. Most discrepancies between systems can be resolved in the interfaces between them. Only in rare cases will significant changes be required to the computational modules in a

system. Since most of the changes will be made in the interface modules, the modularity of these modules contributes more to interoperability than the modularity of the rest of the system.

CONSISTENCY - Software with a high consistency rating (CS.1 and CS.2) will be easier to modify due to its adherence to a consistent set of standards throughout its design and source code. DELETED - see below

SIMPLICITY - If the design structure of the software system is simple, it will be easier for the software engineer to understand, and thus easier to modify. (SI.1) DELETED

SELF-DESCRIPTIVENESS - The quantity and quality of the comments in the source code and the level of the language will directly affect the ability of the software engineer to comprehend the software design well enough to implement the modifications necessary for interoperability. (SD.1, SD.2) DELETED

DELETED: These three criteria, consistency, simplicity and self-descriptiveness were dropped based because they were judged relatively less important than the other adaptability criteria. This opinion was strongly seconded by a large majority of those project managers and software engineers in the interviews. Additionally, we were trying to reduce the large number of proposed criteria to a manageable number.

2.2.3.3 Metrics Measuring the Interoperability of Two Systems

These metrics are based on the concept that, in a particular case, interoperability can only be discussed meaningfully in terms of the two systems that must interoperate. Thus, all these criteria tend to measure how much two systems are alike in those areas required for interoperation. These measurements do not presume that there are ideal characteristics for interoperability; rather they consider only the two systems in isolation and ask: do they work well together in these critical respects?

There are certain philosophical assumptions that are called into question by this approach. The first two metric groups were based on the assumption that there are certain "good"

and "bad" qualities that are predictable in their effect on interoperability. The third metric group ignores these ideals in preference to a pragmatic evaluation of the two systems. This raises a significant question: should the third criterion which compares the two systems without consideration of ideal criteria be used ?

It is possible that two systems individually might be rated poorly using the first two metric groups, but be rated highly interoperable by the third. This could occur when both systems are far from ideal, but are very similar. For example, they might both be written in the same assembly language for the same non-standard hardware using the same non-standard communication interface.

As a result of these considerations, using the three metric groups is proposed to include both idealistic and pragmatic considerations. Using the third set of metrics alone is not recommended. If the system being reviewed has an extended life span, compromises to ideal qualities in favor of short term pragmatic decisions based on similarity to another existing system may prove very costly over the the life span of the software. If this system must eventually interoperate with other future systems, then the price paid for a more 'ideal' system may prove well worthwhile.

The following metrics are included in the third group:

FUNCTIONAL OVERLAP - When there is functional overlap between the two systems, several situations are possible, each of which may adversely affect interoperability. First, one of the overlapped functions may have to be deleted to eliminate the redundancy. When functions overlap, there may be accuracy or timing differences between them which must be resolved to achieve interoperability. Additional complications may result from data dependencies and synchronization problems associated with the overlapped functions. Each of these considerations may involve extra effort to achieve interoperability. There is another possibility; namely, that interoperability will be enhanced by the functional overlap. If the functions are performed identically in both systems, then the effort to achieve interoperation may be reduced. (FO.1 Functional Overlap **NEW**)

COUPLING FACTOR - This metric measures the relative 'closeness' or 'looseness' of the proposed system coupling. This metric measures such aspects as whether the systems share the same hardware, the same data base, the same I/O system, a common operating system, etc. DELETED

DELETED. The idea of the Coupling Factor, so attractive and inventive at first glance, proved to be intractable when applied pragmatically in the real world. None of the researchers who worked on interoperability could come up with a reasonable theoretical framework that could not immediately be disproven with numerous counterexamples. This criteria, in particular, was a principle reason for conducting extensive interviews with personnel whose experience included projects with interoperability problems. These discussions resulted in the same conclusion: 'coupling factor' criteria was unworkable. No one was able to suggest a practical and theoretically sound way of applying it as a criteria of interoperability.

COMMON VOCABULARY - This metric measures the use of the same vocabulary (same words with same meanings) on both projects. Case histories reported in the literature indicates that the usual Tower of Babel differences between projects greatly compounds the problems of achieving interoperation. A third consideration is the effect of the human on interoperability. The report on "NATO Interoperability Handbook of Lessons Learned" cites 'common understanding of terms' as a major problem in achieving interoperability. It should be noted that the handbook is speaking of human understanding, as well as that of computers. Indeed, the human is sometimes the only link between two interoperating systems. (CL.3)

OTHER SYSTEM DOCUMENTATION - The existence and availability of clear, usable, complete and up-to-date documentation of the other system(s) is an important facet of achieving interoperation. (SY.5 Documentation for Other System) NEW

SYSTEM COMPATIBILITY - The following metrics measure the compatibility of the two systems being compared:

DATA COMPATIBILITY - This metric measures the compatibility of data format, code (e.g., ASCII, EBCDIC), access techniques, and security levels. The greater the compatibility of these factors, the greater the potential interoperability. (SY.2 Data Compatibility, part of SYSTEM COMPATIBILITY) NEW

COMMUNICATION COMPATIBILITY - This metric measure the compatibility of communication protocols, transmission rates, and I/O formats. (SY.1 Communication Compatibility part of SYSTEM COMPATIBILITY) NEW

SOFTWARE COMPATIBILITY - This metric measures the compatibility of source language, system software, and utility software. The greater the compatibility of these elements, the greater the potential interoperability of the systems. (SY.4 Software Compatibility part of SYSTEM COMPATIBILITY) NEW

HARDWARE COMPATIBILITY - This metric measures the compatibility of such characteristics as word length, architecture, and interrupt structure, which will impact interoperability. Interoperability will be greatly enhanced if these characteristics are identical. (SY.3

Hardware Compatibility part of SYSTEM COMPATIBILITY) NEW

2.2.4 Tradeoffs Between Interoperability and Other Quality Factors

The goal is to enhance a systems interoperability. The result of achieving the goal is a reduction in the effort required to make the system interoperate with another system. In order to achieve the goal, other quality factors may be sacrificed in the system. Specifically, tradeoffs between system interoperability and other software quality factors must be made. The following factors are defined in McCall (79-1) and are discussed here in terms of tradeoffs with interoperability.

Correctness: Developing new systems to meet interoperability requirements does not adversely impact correctness. However, modifying an existing system may cause inadvertent side affects that degrade correctness. Other correctness sacrifices may result from new problems in synchronization, data sharing and functional sharing.

Efficiency: Whether developing a new system, or modifying an old one, overall efficiency is sacrificed for interoperability. Additional computations may be required to convert character sets, floating point representations and word sizes. Extra checks are necessary to prevent conflicts in accessing and updating shared data structures. Additional resources like buffer space, memory and communication links may be required to support conversion and interface routines. Response time could be increased.

One possible improvement in efficiency may be gained by eliminating redundant operations. By sharing functions, two systems may actually improve overall efficiency. The danger is that by sharing functions, the two systems become dependent upon each other, and may not be capable of operating individually.

Flexibility: Whether modifying an existing system, or developing new systems, flexibility will be partially improved by enhancing interoperability. Both flexibility and interoperability are improved with increased modularity.

However, flexibility will be partially impaired from increased complexity. Generality will be sacrificed at the expense of the additional interfaces required for communications and data commonality. Synchronization constraints will also hinder the ease at which modifications can be made.

Integrity: Improving communications and data commonality generalizes the methods of data access. Generalized access methods result in more difficult access control. The sharing of data by two or more systems requires a larger number of access methods by a potentially larger number of users. Security becomes more difficult, and less reliable. Integrity is sacrificed as interoperability is enhanced.

Maintainability: Whether modifying an existing system or developing new systems, maintainability is partially enhanced by improving interoperability. Both maintainability and interoperability are improved by increasing modularity.

However, maintainability is also partially impaired by improving interoperability. Additional complexity is required to improve communications and data commonality. Complexity hinders maintenance. Other maintenance problems arise from synchronization, data sharing and function sharing, all potential features for increasing interoperability.

Portability: Whether developing new systems or modifying existing systems, portability will be enhanced by improving interoperability. Both portability and interoperability are improved by increasing modularity. The use of I/O standards and protocols will enhance data commonality, communications commonality, software system independence and machine independence. Isolating I/O to a few (or only one) modules will improve both interoperability and portability.

However, interoperability of a system is enhanced by improving its commonality with the other system it will be coupled with. This particular commonality adversely affects portability by making the system both machine and software system dependent.

Reliability: Whether developing a new system, or modifying an old system, reliability is sacrificed for interoperability. Accuracy and precision may be sacrificed for data and communications commonality. Added complexity from synchronization and interface requirements adversely affect reliability. Error tolerances may be altered to accommodate the other system. Inconsistencies may be introduced inadvertently. Synchronization constraints, data sharing and function sharing tend to complicate the system and degrade reliability.

Reusability: Whether developing new systems or modifying existing systems, reusability will be enhanced by improving interoperability. Both reusability and interoperability are improved by increasing modularity. The use of I/O protocols and standards will enhance data and communications commonality, software system independence and machine independence. Isolating I/O to a few (or only one) modules will improve both interoperability and reusability.

However, interoperability is enhanced by improving the commonality of a system with the system it is to be coupled with. This commonality adversely affects machine and

software system independence. In addition, interface and synchronization requirements for enhanced interoperability adversely affect generality. Reusability is degraded as a result of all of these influences.

Testability: Both testability and interoperability are improved with increased modularity. In that regard, testability is enhanced when interoperability is enhanced.

However, improved interoperability means increased complexity for interface, synchronization and data and function sharing requirements. Complexity adversely impacts testability.

Usability: By improving a systems interoperability, no substantial impact results on its usability. Minor affects may occur from increasing data and communications commonality on the ease of interpretation of outputs. The major impact on usability occurs from making the system actually interoperate with another system. Complexity of I/O, response time, accuracy and synchronization problems can all adversely impact the users effort to operate the system.

Table 2.2-4 summarizes the types of tradeoffs and the overall impact of improving interoperability on the other software quality factors. As the table clearly shows, for most quality factors there are both positive and negative affects when system interoperability is improved. However, as the table also shows, the overall impact of improved interoperability on the other quality factors is negative. To develop or modify a system to increase the ease of making it to interoperate with other systems comes at the high price of compromising other desirable software quality factors.

Table 2.2-4 Interoperability Tradeoffs with Other Quality Factors

QUALITY FACTOR	TYPES OF TRADEOFFS	OVERALL IMPACT
Correctness	-	-
Efficiency	+ or -	-
Flexibility	+ or -	-
Integrity	-	-
Maintainability	+ or -	-
Portability	+ or -	+
Reliability	-	-
Reusability	+ or -	+
Testability	+ or -	-
Usability	-	no major impact

Legend: + positive impact
- negative impact

2.2.5 Data Collection

Ideally, different reviewers looking at the same system would provide exactly the same answers to the worksheet questions. The worksheet questions were designed to be as explicit, objective, and non-ambiguous as possible. However, an extensive human factor evaluation was not conducted.

All the interoperability worksheets were completed by the same person. This should assure consistent interpretation of worksheet questions, but it does raise the question of how the researcher's own biases may have skewed the data for all projects. The specific nature of the questions are designed to minimize this effect. The bias, if it occurs, is most likely on Worksheets 1 and 2, because the questions are comparatively more broad and subjective.

The data from the three projects is summarized below. As described above, the modules from each system were identified by the project as being impacted principly by interoperability considerations.

	DATA EXAMINED		
	SYSTEM A	SYSTEM B	SYSTEM C
NO. OF MODULES	57	11	18
LINES OF CODE	5253	1937	6122

Not all of the modules in the impacted portion of these systems were examined. Representative sample modules were examined, and the table above describes those selected, not the total.

2.2.5.1 Data Collection Methodology

Data was collected from three separate projects which all had interoperability requirements or considerations. Specific software modules were selected from each project based on the recommendations of project personnel. Those modules were changed or written due to interoperability considerations. For example, the Project C Communication software was chosen because it had specified interoperability requirements. The Inertial

Navigation System of Project B software was identified as the software most impacted by interoperability considerations. Finally, Project A software was a deliverable product that generated mission data and it had to interoperate with several entirely separate systems.

System level data for Worksheets 1 and 2 were gathered by reviewing high level specifications that would have been available at the time of normal completion of these worksheets. This approach was supplemented by interviews with project personnel to confirm our understanding of the specifications. We attempted to complete the forms using the knowledge that was available at the appropriate time in the project history, rather than using current specifications.

Worksheets 3 and 4 were completed using source code listings and the output of support programs. Some of these support programs easily provided details that entailed tedious manual techniques on other projects which did have equivalent support software. Project A, for example, had extensive set/use and call linkage tables which provided easy answers for some Worksheet 3 and 4 questions.

2.2.5.2 Comments on Data Collection and Metric Computation

Numerous anomalies arose during the evaluation of collected data and the computation of metrics. Certain metrics in the framework which appeared simple, easy to apply, and virtually foolproof proved difficult to measure, or unrealistic to compute as originally defined, or, in the worst cast, meaningless in context.

An example of the latter is the computation of modularity used in the evaluation of interoperability. The computation is based on (in part) the ratio of various calling sequence parameters to the total number of calling sequence parameters. The computation of modularity includes the ratio of the number of control parameters to the total number of parameters in the calling sequence list. On one of the projects surveyed, there were no calling sequence lists because the program design stored all data in common storage areas. The design approach was used to reduce the storage space that would have been required by the use of calling sequence linkages.

How, then, should the modularity of a module which does not use calling sequence parameters be rated? Should it be arbitrarily rated '1' because it doesn't have any calling sequence parameters? Or should it be rated '0'? In the case under consideration, modularity is, perhaps, a much more complicated function of the set/use patterns of variables in the common storage areas. It is possible to devise metrics that would measure the use of variables in common storage areas by each module; however, the data would have to be extracted by a set/use utility, which the project in question did not have. Before developing such an elaborate metric, however, a new question arises: Is it worth developing this intricate and expensive-to-apply metric to measure modularity (as redefined)? In the present case, the project had no need for such support software, and it was too costly to develop as part of this research. The modules were rated '0'.

At first, questions on a second system that was unknown at the time of the worksheet's theoretical completion resulted in a response of N/A (not applicable) on the worksheets. This approach was later re-evaluated: if the information was not available, then that fact should be counted against the system being evaluated. The presence of second system data should ideally add to the potential for interoperation; therefore its absence should subtract from its interoperability rating. All N/A's were then changed to 0's.

2.2.6 Metric Validation

The discussion of metrics validation is divided into two sections: the first describes how interoperability ratings were derived, the second reviews the results of the validation.

2.2.6.1 Selection of Interoperability Ratings

In order to validate the metric framework for interoperability, it is necessary to find a measure of effort for each of the modules from the three projects. This proved problematical. None of the projects considered for interoperability had collected data on the effort to develop individual modules. Project-level productivity rates were available on two of the projects in terms of lines of source code per man-month. These figures did not permit reasonable comparison of modules, since if the rate was constant on a project, the effort spend on each module was directly proportional to the number of lines of source code in each. This meant we were comparing the metrics not to a measure of

interoperability, not to a measure of effort, but to the size of the module. Out of curiosity, this comparison was made and the results were not meaningful.

The next consideration tried to compare the effort between projects. Each of the projects had a productivity rate used for estimating software development costs. The productivity rates reflected very different contractual requirements and software development environments, so these rates were not comparable and could not be used for developing a rating for interoperability effort.

Four other approaches were then considered for developing an interoperability rating factor for validating the metrics.

The first approach was to consider the ratio of effort to achieve interoperability to the effort to develop a completely separate and independent interface software product that would couple the two systems together. This approach would require the project personnel to estimate the effort used to achieve interoperability in reality, and to also estimate the effort to build a completely independent software interface product rather than modifying the system under study. This approach was rejected because the second estimate was extremely subjective when compared to the approach finally selected.

The second approach was to perform a Delphi survey* on the comparative interoperability of the three projects included in the study. That is, to interview a number of 'oracles' who had sufficient knowledge of all the systems to assess their relative success at achieving interoperability. This approach was rejected because we couldn't locate enough oracles with the first-hand knowledge of all three systems could not be located. This approach also seemed too subjective.

The third approach was to consider the ratio of software to hardware costs entailed to achieve interoperability on each system. The argument was that the greater the relative proportion of software cost (to hardware cost), the greater the effort to achieve interoperability. Several contradictory examples were raised and this approach was dismissed.

* Lindstone, Harold A. and Murray Turoff. (Editors), "The Delphi Method-Techniques and Applications", Addison-Wesley, Reading, Mass., 1975

The fourth approach was to consider the ratio of the effort spent in achieving interoperability to the effort to initially develop the software system.

Rating = 1 - eff2/(eff1 + eff2)

where eff1 = effort to build the software without interoperability (or initial effort)

eff2 = effort to achieve interoperability

The three effort ratings were

Project A	0.75
Project B	0.61
Project C	0.91

As noted above, it was problematical to isolate the interoperability costs from the costs of other enhancements and refinements that were included in the budgets during the effort to achieve interoperability. However, there were people on each of the three projects that had worked during both the development and interoperability phases; the ratios of effort were determined from interviews with these personnel. While these ratios are also subjective, they are based on intimate knowledge of the project histories; these people had a sense of what proportion of the work was due to enhancements, and what proportion was due to interoperability considerations. This approach was found to be the most attractive and reliable, and it was implemented. All interoperability rating factors were derived in this way.

2.2.6.2 Validation Results

Table 2.2-5 Summarizes the metric scores for each System (A, B, C). Tables 2.2-6 and 2.2-7 summarize the composite criteria and metric scores for all the three projects.

Table 2.2-5 Interoperability Metric Summary (by Project)

WORKSHEET	METRIC	PROJECT		
		A	B	C
1	CL.1	.5	.92	1.0
	CL.2	1.0	0.0	1.0
	CL.3	0.0	N/A	0.0
	CM.2	0.0	.33	.5
	SY.5	0.0	N/A	0.0
	FO.1	.75	N/A	0.21
	Average	.37	.42	.62
2	CL.1	.33	1.0	.62
	CL.2	.69	1.0	.70
	CM.1	.46	.69	.83
	CM.2	.74	.60	.76
	SY.1	N/A	.50	.25
	SY.2	0.0	0.0	0.0
	SY.3	.33	0.0	0.0
	SY.4	0.0	.33	0
	Average	.36	.52	.40
3	AG.1	.28	.42	.43
	AG.2	0.0	0.0	.33
	ID.1	.98	1.0	1.0
	ID.2	.99	1.0	1.0
	MO.2	.78	.85	1.0
	MO.2*	.61	.65	.75
	Average	.61	.65	.75

* = Repeated measurement during test phase

Table 2.2-6 Interoperability Metric Summary (by criteria)

CRITERIA	A	B	C	MEAN	STD.DEV.
Commonality	.58	.66	.58	.58	.40
Communicativeness	.3	.49	.62	.54	.24
System Compatibility	.33	.21	.06	.20	.29
Augmentability	.14	.21	.38	.24	.18
Independence	.99	1.0	1.0	.99	.01
Functional Overlap	.75	0	.21	.32	.39
Modularity	.71	.75	.88	.76	.58
TOTAL SCORE	.53	.47	.53		

Table 2.2-7 Interoperability Metric Summary

METRIC	RANGE	MEAN	STD.DEV.
cl.1	.33 - 1	.73	.26
cl.2	0 - 1	.73	.35
cl.3	0 - 0	0	0
cm.1	.46 - .83	.66	.19
cm.2	0 - .76	.49	.26
sy.1	.25 - 1	.58	.38
sy.2	0 - 0	0	0
sy.3	0 - .33	.11	.19
sy.4	0 - .33	.11	.19
sy.5	0	0	0
ag.1	.28 - .43	.38	.08
ag.2	0 - .33	.11	.19
id.1	.98 - 1	.99	.01
id.2	.99 - 1	.99	.006
mo.2	.05 - 1	.76	.58
fo.1	0 - .75	.32	.39

For the purposes of exploratory analysis, 'median polish' analysis on the worksheet scores was performed. This method summarizes patterns in a table of medians (Table 2.2-8). Medians as opposed to means are used because medians are a more resistant measure of location. In other words, a mean from a sample containing an outlying data point would be skewed by that extreme value; whereas a median would not be as greatly affected. The median is more descriptive.

Table 2.2-8 Median Metric Scores by Project

	PROJECT		
WORKSHEET	A	B	C
1	.25	.33	.75
2	.33	.55	.43
3	.69	.75	.87
Rating	.75	.61	.91

From this matrix of medians, the residual median variability in each cell was computed. Then a value R was computed, this is a measure of the percentage of cell median variability explained by the differences between worksheets and the differences between projects. This value was 62%, which implies that 38% of the variability comes from effects other than the project-to-project differences and worksheet-to-worksheet differences. These effects might be errors on worksheets, human transcription errors, or conceptual errors in the metric framework.

Since the worksheet 2 score of project C was the most anomalous, it was changed in the above analysis from .435 to .8, so that it would lie between system C's worksheet 1 and 3 scores and the analysis was repeated. This changed the computed R from 62% to 69%, which indicated that even if the worksheet 2 value of .8 is correct, a fairly large percentage (31%) of variability from unknown sources still remains. This was not an encouraging sign for the validation process.

Several different analytic approaches were used to validate the data. Using the interoperability rating factors, a sequence of equations were set up in the form

$$R = ax1 + bx2 + cx3$$

where

x1 = unweighted mean metric score from Worksheet 1

x2 = unweighted mean metric score from Worksheet 2

x3 = unweighted mean metric score from Worksheet 3

One such equation was written corresponding to each of the three systems, resulting in a system of three equations and three unknowns, which could then be solved using standard mathematical techniques. The results were

$$a = -0.68$$
$$b = -1.59$$
$$c = 2.62$$

These results are counter-intuitive; the negative coefficients for the first two worksheets might be interpreted to mean that metrics from Worksheets 1 and 2 were negatively correlated with interoperability. At this point several other ways of considering the worksheets were explored; worksheet 3 was eliminated, worksheets 1 and 2 scores were averaged and used with worksheet 3, and the 'second system' metrics were removed from worksheet 1 and 2 scores. Additionally, various combinations of the above approaches were used. In each case, when the resulting set of equations were solved, the coefficients varied widely. Due to the small number of cases (systems), the mathematical techniques were very sensitive to the odd sequence of scores on System C. That is, because System C scored higher on worksheet 1 than on worksheet 2, only the worksheets 1 and 3 scores correlated closely to the ratings.

Since the interoperability ratings had been determined by using a modified Delphi method (interviews, not worksheets), there was considerable margin for error in these ratings. So we analyzed the sensitivity of the equations to errors in the ratings was analyzed; that is, the effect on the solution (a,b,c) due to changes in the ratings of systems A and B was determined. Using the results of this analysis, the limits of errors in ratings for systems A and B which would result in all positive coefficients were computed.*

* This analysis is a modification of the Sensitivity Analysis described in section 3 of chapter 8 of "Linear Programming" by Saul I. Gass, McGraw Hill, N.Y., 1975.

When a system of linear equations

$$Ax = b$$

is solved, yielding

$$x_o = A^{-1}b$$

some of the components of x_o may have intuitively objectionable characteristics, such as being negative when their interpretation excludes such values. That is, $x_o \quad 0$ is required.

One possible remedy is to assume that the right hand side (r.h.s) vector b contains errors in some of its components. To analyze the sensitivity of the system to such errors, the vector b may be incremented by a vector d, producing a hypothetically correct r.h.s. vector b'. When the resulting system is solved, there follows:

$$\begin{aligned} Ax &= b + d = b' \\ x' &= A^{-1}b' \\ &= A^{-1}b + A^{-1}d \\ &= x_o + A^{-1}d \end{aligned}$$

The requirement $x' \geq 0$

produces a set of constraints on the components of d; they must satisfy the relation

$$A^{-1}d + x_o \geq 0$$

Analysis of these constraints may produce a set of feasible d components, the magnitude of which could qualify them to cancel errors in the original b components.

Using the actual unweighted mean of the metric scores, we have

$$A = \begin{bmatrix} .37 & .36 & .61 \\ .42 & .52 & .65 \\ .62 & .40 & .75 \end{bmatrix}$$

the interoperability ratings

$$b = \begin{bmatrix} .75 \\ .61 \\ .91 \end{bmatrix}$$

it is found that

$$x_o = \begin{bmatrix} -.681 \\ -1.59 \\ 2.62 \end{bmatrix}$$

The first two components of x_o are negative, contrary to requirements. To test the sensitivity of the solution to small changes in these components, define

$$d = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

and set $Ax = b'$, where $b' = b + d$.

Since

$$A^{-1} = \begin{bmatrix} -10.64 & 2.62 & 6.30 \\ -7.66 & 7.87 & -.66 \\ 12.83 & -6.35 & -3.50 \end{bmatrix}$$

the conditions on the d components to produce a satisfactory solution

$$x' = A^{-1}d + x_o \geq 0$$

are these:

$$x_1' = -10.64\, d_1 + 2.62\, d_2 + 6.30 d_3 - .68 \geq 0 \qquad \text{I}$$

$$x_2' = -7.66\, d_1 + 7.87\, d_2 - .66 d_3 - 1.59 \geq 0 \qquad \text{II}$$

$$x_3' = 12.83\, d_1 - 6.35\, d_2 - 3.50 d_3 + 2.62 \geq 0 \qquad \text{III}$$

Some insight into an appropriate choice for the components of d is gained by forming the partial derivatives of the components of x' with respect to the components of d. Note that the partial derivative of x_i' with respect to d_j is the (i, j)-th element of A^{-1} and that the partial derivative of x' with respect to d_j is column j of A^{-1}.

The interpretation of $\frac{x'}{d_3}$ is the rate of change of the solution vector x' with respect to d_j alone. In particular, consider

$$\frac{x'}{d_3} = \begin{bmatrix} 6.30 \\ -.66 \\ -3.50 \end{bmatrix}$$

and assume that initially $x' = x_o$, with $d = \underline{0}$. Then if d_3 increases, with d_1 and d_2 held fixed, x_1' will increase, as desired, but x_2' will decrease, becoming even more negative. Similarly, if d_3 decreases, again with d_1 and d_2 held fixed, x_2' will increase, as desired, but x_1' will now decrease, becoming more negative.

That is to say, there is no change in d_3 which causes an improvement in both x_1' and x_2' together. An improvement in one is made to the detriment of the other. Therefore, this component of d will not be varied:

$$d_3 = 0$$

The set of feasible d_1, d_2 pairs which satisfy inequalities I-III form a triangular region in the d_1, d_2 plane. The additional constraints

$$0 \leq b' \leq \underline{1}$$

trim down the region of feasibility slightly:

$$0 \leq .752\, d_1 \leq 1$$

$$0 \leq .606\, d_2 \leq 1$$

Any point of the feasible region can be used as a test of sensitivity of the system; e.g., $d_1 = 0$, $d_2 = .3$ gives

$$b' = \begin{bmatrix} .75 \\ .91 \\ .91 \end{bmatrix} \text{ and } \quad x' = \begin{bmatrix} .11 \\ .77 \\ .71 \end{bmatrix}$$

Similarly, the vertex $d_1 = -.2$, $d_2 = .007$ produces

$$b' = \begin{bmatrix} .55 \\ .61 \\ .91 \end{bmatrix} \text{ and } \quad x' = \begin{bmatrix} 1.47 \\ 0 \\ 0 \end{bmatrix}$$

That is, seemingly small changes in the r.h.s. vector b produce major changes in the components of the solution vector.

This shows that expected errors in the ratings would have a large effect on the solution. The system of equations is very sensitive to error in the rating factors, probably due to the very small number of cases available for analysis. Thus, no solution can be accepted with any degree of confidence.

Several additional approaches were tried. It seems reasonable that at each phase in the development cycle when interoperability is assessed with metrics, that the results of previous assessments be used in conjunction with the current assessment. One would expect to see the interoperability assessment converge toward a realistic prediction as additional knowledge was gained. The scores at each phase were averaged with previous scores; that is, at phase 2 the mean score from worksheets 1 and 2 was computed, at phase 3, the mean score from worksheets 1, 2, and 3 was computed (see Table 2.2-9). This, indeed, tended to smooth out the predictions, but not enough to significantly reduce the sensitivity of the system of equations to small rating errors.

Table 2.2-9 Mean Cumulative Worksheet Scores

	PROJECT A	PROJECT B	PROJECT C
mean score 1	0.37	0.42	0.62
mean score 2	0.37	0.47	0.51
mean score 3	0.44	0.53	0.59
Q. Rating	.75	.61	.91

This approach gave the following ratings to the worksheets

$$x' = \begin{bmatrix} -.68 \\ -1.6 \\ 2.6 \end{bmatrix}$$

A third approach was to average worksheets 1 and 2 and to retain worksheet 3. This results in

$$x' = \begin{bmatrix} -2.6 \\ 2.9 \end{bmatrix}$$

Since visual inspection of the metric scores for cl.3, sy.5, and fo.1 on worksheet 1 and sy.1, sy.2, sy.3, and sy.4 indicates they may be problematical, we excluded them from the worksheet score. This resulted in

$$x = \begin{bmatrix} 0.58 \\ -0.89 \\ 2.0 \end{bmatrix}$$

Once again, the solutions were shown to be very sensitive to change of approach, and none of the approaches showed any great promise.

The significance of individual metrics was also considered. It was conjectured that the last measured value (most recent) of each metric be used, since the last sample estimate of each metric is presumed to be the most accurate one. These are shown in Table 2.2-10. CM.2, the second commonality metric, was the only single metric that vaguely corresponded to the interoperability ratings.

Considering various combinations of metrics, there are several combinations of "last sample" metrics which seem quite descriptive. These combinations are:

1. CL.1, CL.2, FO.1 (user input and user output effectiveness, and functional overlap)
2. CL.2, CM.2, FO.1 (user output effectiveness, data commonality, and functional overlap)
3. CL.2, CM.2, FO.1, MO.2 (same as (2) above plus modularity)

Their relationship to the estimated interoperability are shown in Figure 2.2-3. It should be remembered that these plots should be considered as descriptive, not predictive. There is simply not enough projects (there are only 3) to conclude that there is predictive significance to these plots.

Table 2.2-10 Last Sample Metric Values

Last Value	A	B	C
CL.1	.33	1.0	.62
CL.2	.69	1.0	.70
CL.3	0	0	0
CM.1	.46	.69	.83
CM.2	.74	.6	.76
SY.1	0	.5	.25
SY.2	0	0	0
SY.3	.33	0	0
SY.4	0	0	0
SY.5	0	0	0
FO.1	.75	0	1
AG.1	.28	.42	.43
AG.2	0	0	.33
ID.1	.98	1.0	1.0
ID.2	.99	1.0	1.0
MO.2	.61	.65	.75
Q. Rating	.75	.61	.91

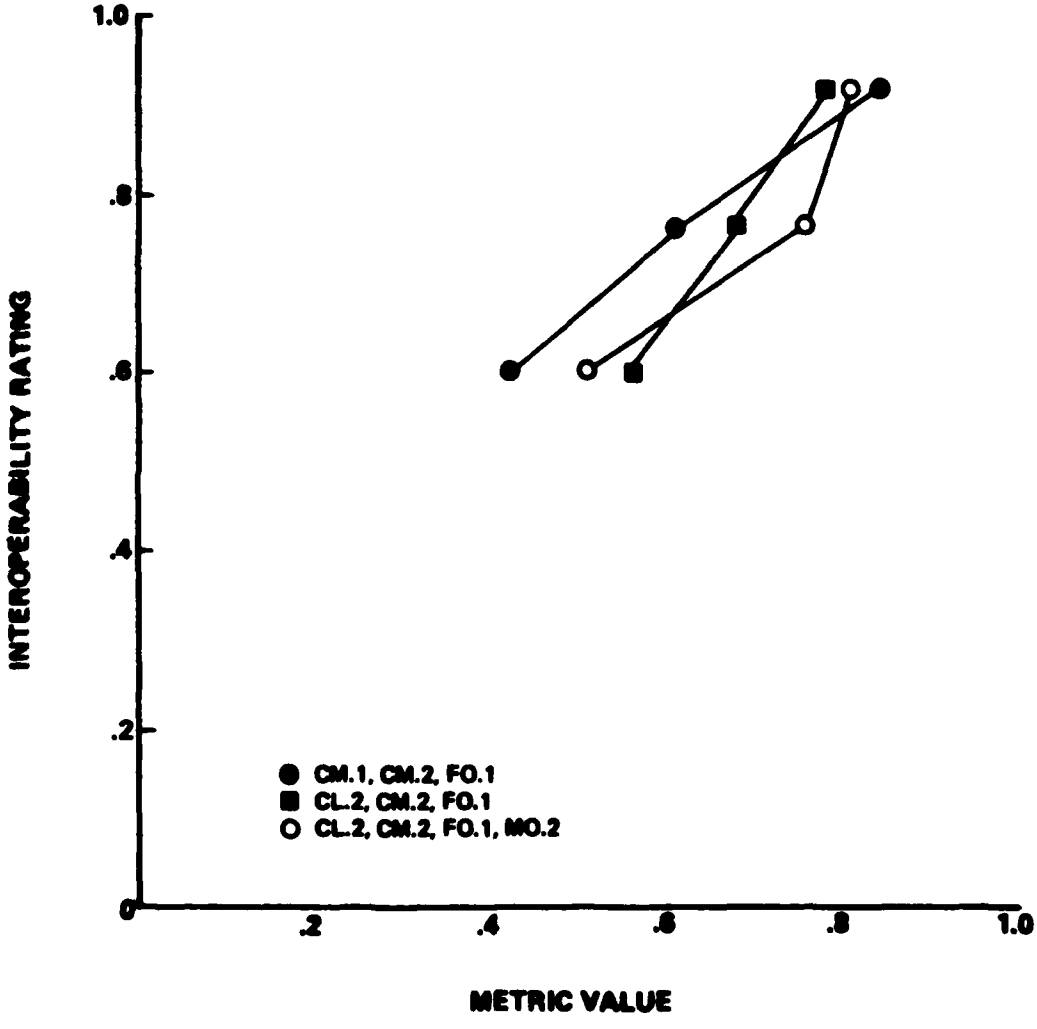


Figure 2.2-3 Metric Combinations vs. Interoperability Ratings

2.2.7 Conclusions and Recommendations

This study produced several conclusions on the subject of measurement and prediction of interoperability. These conclusions are interrelated, and so will be discussed together. In order that the recommendations can be understood in context, they are presented in the discussion along with the conclusions which prompted them.

Interoperability is a system level measure only, which means that we cannot speak meaningfully of the interoperability of a software module. We can only speak of the interoperability of the embedded software system as a whole. This means that we can only measure interoperability at the system level. The operation definition of interoperability used in this metric research is consistent with this interpretation.

2.2.7.1 Definition of Interoperability

The working definition was chosen by the original researchers because it gave them something they could measure: most projects have some estimate of manpower expended. This, in turn, provided a measure against which the metrics could be correlated. This approach was attractive, because it would provide a method to predict cost.

An unfortunate side effect of this definition is that there is no measure of how well the systems couple together, all that is evaluated is how much it cost to achieve the coupling. Only costs (effort) are evaluated when the interoperability of two systems is compared, the relative success of the two interoperability efforts is not evaluated. This may be the only workable definition at the moment, but it is a poor one. Is effort what should be measured? Although effort is important because it is a measure of cost, the constrained definition omits any measure of how well two systems work together.

A more realistic definition of interoperability must be developed, otherwise the metrics will measure something quite different from interoperability. For example, if effort is used as the measure of interoperability, on a project where effort is available only as a general relation of lines of code/man-month (which includes documentation, reviews, etc.), the effort figure that is used will be directly related only to module size. Thus, metrics are being correlated not to interoperability, not even to effort, but to module size. Is module size really directly equivalent to interoperability? Very unlikely.

The major problem in the validation of interoperability metrics was the inability to get a good measure of the interoperability of the software modules evaluated. Without that measure, there is nothing which the metrics can be correlated against to evaluate their success or failure.

The definition of interoperability as a measure of effort to achieve interoperability may be a dead end. If reliability had been defined as the effort required to achieve a reliable product, it is highly unlikely that the field of reliability would have developed into a practical and useful science. Similarly, interoperability will never be an analytically useful field of study until it is defined in a quantitative way. There are several possible interoperability definitions that may be worth consideration. These alternative definitions use factors other than effort in the evaluation of interoperability. Each has its advantages and disadvantages, which are discussed below.

The first definition is the fraction of total functions shared with respect to the total of all functions on both systems. This definition measures how much of each system is mutually used by the other, which gives some measure of how much functionality is shared. The disadvantage of this approach is that the measure becomes a relatively subjective one. One expert observer may disagree with another, because of varying interpretations of the words "function" and "shared." Care would be required to define these terms so that they are unambiguous, easy-to-understand, and easy-to-apply.

A second definition is the fraction of data shared mutually with respect to the total data of the two systems. This approach has the advantage of being more easy to measure objectively, since most observers can agree on whether certain data is used mutually or not. The disadvantage of this definition is that is does not consider how the data is used functionally. One function may require a very large data base, another only a few words of storage; yet the second function may be more critical to interoperation of the two systems.

If there appears to be no way to quantify interoperability, and if effort to achieve interoperability remains difficult to measure; interoperability may be beyond the ability of current technology to measure or predict. The system-level considerations inherent to this quality factor may put it beyond the ability of the metrics approach to predict.

2.2.7.2 Recommendations for Data Collection

Interoperability metrics require information not easily available from normal CDRL documentation, and not normally collected as part of the development process. With the information normally available to the contract's technical monitor and the procurement agency, it is very difficult, if not impossible, to compute interoperability metrics. Therefore the procurement agency will require additional information from the developer. This additional information should be obtained by making it a contractual requirement via the CDRL. Metric data collection procedures must be designed to reduce, to the greatest extent possible, any subjective evaluation by the reviewer. Therefore, the metrics must be based on the most accurate information available; this is only known by the developer. New CDRL items or revised Data Item Description forms (DIDs) must be developed. For example, the DIDs for Preliminary and Critical Design Reviews could include requirements for metric data.

The procuring agency could, alternatively, require the contractor to compute the required metrics, and to deliver these as part of CDRL items during various phases of the contract. If this alternative is chosen, it would be prudent to require that the data used to derive the metrics also be delivered.

The primary decisions that affect interoperability are made during the earliest phases of requirements analysis and definition. The requirements should include very precise interface definitions. Most of the pertinent questions can be answered only by the project designers. If interoperability metrics are imposed on a contractor, the data necessary for computing the metrics must be specified in CDRL requirements.

Even the assumption that the major cost of modifying a software item is due to its interoperability is problematical. Many different factors affect the cost of modification. Even when specific data are available on the cost to modify a specific system, and the reason for that modification is supposedly only to achieve interoperability with another system, this assumption is suspect. Many changes are made for reasons that have nothing to do with interoperability: to restructure the program more efficiently, to improve its maintainability, to correct deficiencies. These changes are often not tracked in the overall budget because many of the changes have been informally requested by the

customer, or are implemented as a no cost change. Therefore, projects should be chosen for validation only when explicit interoperability effort is available.

To conclude, it is very difficult for the Contract Technical Monitor or Acquistion Manager to determine the answers necessary to compute interoperability metrics without requiring additional data from the developer. It will be important to assess the cost effectiveness of this additional burden. It appears that the most meaningful data is collected during the early phases of requirements analysis and definition.

2.2.7.3 Number of Projects Required for Validation

This study located three projects with interoperability requirements that had to be solved, and which had data available for the computation of the metrics. This means that only three interoperability ratings were available (one for each system) for the validation of the proposed interoperability metrics. So no matter how much metric data is collected on the three projects, there are only three interoperability ratings against which to validate them. Mathematically, this is not adequate; which leads to the next conclusion -- there must be a reasonably large number of projects in order to validate interoperability metrics. Certainly, at least eight; ideally, a dozen or more.

2.2.7.4 Future Research

The current metric framework assumes that all the metrics are orthogonal; that is, none of the metrics are measuring the same thing. No one, however, has proven this to be true. No one has demonstrated that one metric is not, for example, a linear combination of other metrics. Factor Analysis could be used as an approach to demonstrating that the metrics are orthogonal.

The minimum number of metrics should be used for a specific criteria. If three metrics can be shown to provide an estimate that correlates to the effort required to achieve interoperability as well as a five-metric measure, then the three metric measure is clearly preferable. Various combinations of metrics could be evaluated, and the estimate should be chosen based on the smallest number of metrics and that has an acceptable residual error. This approach will help reduce the cost of implementing software metrics.

The analysis reported in section 2.2.6 suggests that the criteria of Communicativeness, Commonality, Functional Overlap, and Modularity are most closely descriptive of the interoperability scores of the three projects studied. More specifically, the metrics CL.2 (user output effectiveness), CM.2 (data commonality), FO.1 (functional overlap), and MO.2 (modularity) seem to provide a close description of the scores. The word 'description' is used because the data and analysis are not conclusive enough to warrant their use as predictors. These metrics and criteria appear to be the most promising, and should be used as the basis for further research. A second consideration is the cost to implement and administer such a system of metrics. Not only should such a metric system be validated, but it should also be cost effective. The cost effectiveness of measuring and predicting interoperability by software metrics should be evaluated by a phased implementation over several projects. If the use of metrics to predict interoperability is judged cost effective, then it is feasible to begin phased imposition of interoperability requirements. Again, the cost effectiveness of these requirements will require careful evaluation.

2.3 REUSABILITY

Software reusability is defined in the Software Quality Measurement Manual (McCall 79-1) as the "extent to which a program can be used in other applications", and is measured in terms of "the effort required to move a program, or a part of a program, to another application". Reusability can also be defined in the sense of "the potential of the software for reuse". Reusability is a measure of the extent to which a software program can be used in other applications, related to the packaging and scope of functions that programs perform, either in part or in total. If the effort required to reuse the software is much less than that required to implement it initially, and the effort is small in absolute sense, then the software program is highly reusable. Note that this does not exclude complete rewriting of the reusable program modules. The degree of software reusability is determined by the number, extent and complexity of the changes, and hence by the difficulty in software reusable implementation process. A universal reusable software program can be considered as a completely portable software program.

Software consists of the programs and documentation which result from a software development process. In order to evaluate software in a systematic manner it is important to understand what quality or qualities are most important from user's point of view. In this study, we present a method of associating the generally understood software qualities to more specific and measurable software characteristics which become the basis for the evaluation. In order to bridge the rather large gap between user understanding and software itself an intermidate set of user-oriented software quality criteria are defined. The hierarchy is illustrated by Figure 2.3-1.

The individual metric score is accumulated to obtain the criteria score, and each criterion score is accumulated to obtain the overall score for the quality factor—software reusability. The evaluation for the software reusability is designed and presented in the metric worksheet. The actual evaluation procedure take the following steps: collect the worksheet data for the project, compute the metric score from raw worksheet data, and finally perform regression analysis on the reusability rating and computed metric data.

It should be noted that not all the software quality factors are complementary with software reusability. For example, the very characteristics which in a given application

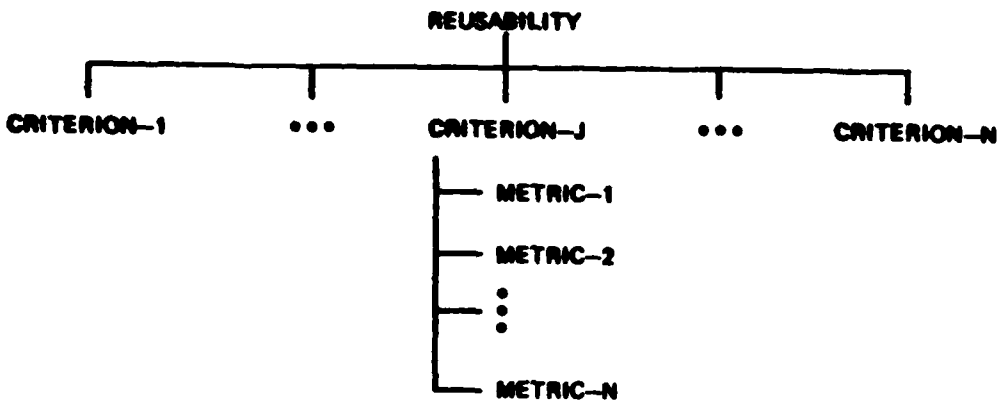


Figure 2.3-1. Reusability Hierarchy

have a positive effect on reusability may well have a negative effect on efficiency. This fact is due in part to the current state-of-art in hardware architecture versus software architecture is that there are usually no directly efficient implementations of high order language constructs. Thus, there is the commonly held view that assembly language is the necessary source language for real-time programs which require high effiency.

2.3.1 Key Concepts

Three key concepts which determine the reusability of a software product can be identified. These are level of reuse, extent of reuse, and degree of reuse. These concepts and their subcategories are shown in Table 2.3-1.

TABLE 2.3-1 Key Concepts of Software Reusability

REUSABILITY CONCEPT	SUBCATEGORY OF CONCEPT
LEVEL of REUSE	Module Functional System
EXTENT of REUSE	Partial Total
DEGREE of REUSE	Low Medium High

There are three distinct levels of software reusability: module, functional, and system. Module reusability refers to individual modules which can be reused in different systems. A subroutine that computes the "square root" function exhibits module reusability. Functional reusability refers to groups of modules which together perform a specific function or related functions, which can be reused in other systems. Navigation

subsystem and mathematical libraries exhibit functional reusability. System reusability entails the reuse of the entire software package, such as spares inventory program and DAIS (Digitized Avionics Informations System) program.

Software reusability can also be viewed in terms of the extent of reusability: partial or total. Partial reusability is exhibited when software can be reused in another application with some modifications. These modifications may consist of changing only a portion of the executable code. An example of this type of reuse is in an operating system device driver. Rarely is a new driver written from scratch. Instead, a driver for a different device is usually modified to handle the new device. Total reusability implies reuse of the whole software system, with only those changes which are necessary to make it run in a new environment. The functions, interfaces, etc., of the system are left essentially the same. This type of reuse is examplified by the "square root" subroutine that is reused, unchanged, in several applications.

The degree of reuse of a software product relates to the items that can be reused from each phase of development. Generally, the more abstract an item is, the more reusable it is. For example, a concept is more reusable than a design, which is more reusable than the code. This is because at each step in the development, decisions must be made which tend to limit the reusability of the item. For example, the selection of any language to implement some software will diminish its reusability because someone may need the software in another language, although the design may be reused. On the other hand, the more abstract an item is, the more work is required to produce a useful product from it. In general, a high degree of reuse means that more concrete items, such as code, test procedures, etc., may be reused with minimal effort. A low degree of reuse would signify that only fairly abstract items, such as concepts or algorithms, may reasonably be reused.

There are exceptions to this general rule. For example, a module may be directly reusable in a new application, but the requirements, design, integration and acceptance documentation and test cases may not be suitable for reuse. In this case the code which represents 15% of the cost would be saved, the other 85% would not be avoided.

On the other hand the requirements, design, integration and acceptance documentation and test cases may be reused, but the module itself may have to be recoded. An example

is the replacement of a discrete Fourier Transform with a Fast Fourier Transform. In this example the code is not reusable but 85% of the cost is saved.

2.3.2 System Characteristics

This section identifies system characteristics which impact the reusability of software. Table 2.3-2 details all the system characteristics that were considered with respect to reusability. Those marked "accepted" are defined in section 2.3.2.1 and 2.3.2.2 and described in terms of their impact on software reusability in section 2.3.2.3. Those marked "rejected" were considered to be redundant or of less importance in their impact on reusability, and were not investigated.

TABLE 2.3-2 System Characteristics for Reusability

SYSTEM CHARACTERISTICS	IMPACT ON REUSE	
	ACCEPT	REJECT
Accessibility	x	
Scope of Functions		
Specificity	x	
Commonality	x	
Completeness	x	
Independence		
Data independence	x	
Machine independence	x	
Software system independence	x	
Standard computer architecture	x	
Microcode independence	x	
Word size		x
Memory addressing limitations	x	
Data Management Methodologies		
Database management systems		x
Access methods		x
Mass storage devices		x
Data Timing		x
Algorithms	x	

TABLE 2.3-2 (Continued)

SYSTEM CHARACTERISTICS	IMPACT ON REUSE	
	ACCEPT	REJECT
Languages		
Acceptability	x	
Level (Order)		x
Ease of use		x
External Control Modes		x
Fault Tolerance	x	
Hardware		
Analog		x
Digital		x
Documentation		
Completeness	x	
Correctness		x
Organization	x	
Data Structures		
Complexity		x
Parameterization (extensibility)	x	
System Complexity		
Interface complexity	x	
Module coupling	x	
Output Mode		x
Security		x
System Architecture		x
System Availability		x
System Maturity		x
System Reliability		x
System Size		x
System Sophistication		x
Timing		x

2.3.2.1 Applicability of Interoperability Characteristics to Reusability

The following definitions are those interoperability characteristics which have applicability to reusability.

INDEPENDENCE:

Machine Independence	- Software not reliance on unique characteristics of a specific machine.
Software System Independence	- Application software not on utility and features of a specific operating system.
Algorithms	- A plan consisting of a series of operations to achive the desired result.
Languages-Acceptability	- Systems of symbols which can be used to provide instructions to computer and acceptable to the system requirement.
Fault Tolerance	- The ability to detect, contain, diagnose and recover from faults.
Documentation-Completeness -Organization	- Completed written and well-organized forms or records to aid in the operation and explanation of a system.
Data Structures Parameterization	- A parameterized data structure to allow organized access and storage of data (e.g. lists, trees).

2.3.2.2 Identification of New Characteristics

The characteristics defined in this section are also important in the analysis of tne software quality of reusability.

Accessibility	- The ease of access to system source code and documentation.
Scope of Functions:	
Specificity	- The degree to which all of the modules in a system perform single, precisely defined functions.
Commonality	- The usefulness, to other applications, of the functions performed by the software.
Completeness	- The degree to which a system performs a total function (in terms of user needs).
Independence:	
Data Independence	- Those attributes of a software system that make necessary data usable in all the environments (e.g. location independent data in a tracking system).
Standard Computer Architecture	- The use of a standard computer architecture or instructure set for all applications.
Microcode Independence	- A very low level machine language is not used to implement conventional machine language instructions.
Memory Address Limitations	- The upper limit on the amount of memory that a computer can directly access.
System Complexity:	
Interface Complexity	- The ease of understanding module interfaces, in terms of the data affected.
Module Coupling	- The level of interdependence between modules within a system.

2.3.2.3 Impact of Characteristics on Reusability

The characteristics of software systems have a significant impact on the effort required to reuse them. The following paragraphs discuss the relationships, where they exist, between characteristics of software and its reusability.

2.3.2.3.1 Accessibility

Ease of access is a prime determinant of the reusability of software. There are thousands of potentially reusable software products that are not utilized because subsequent designers do not know of their existence or they are too difficult to obtain. In light of this, it is probably better to have some moderately reusable products gathered in one place, than to have many highly reusable software products which are inaccessible or unknown to potential users. The term "software product" is used here to denote code, design documents, requirement documents, test procedures, etc. A system designer will choose to build a new item rather than reuse an existing one unless the expected effort to obtain and adapt the reusable product is considerably less than the expected development effort for a new one (Bowen 81). If a designer must spend several days or weeks tracking down and evaluating software for possible reuse, he is unlikely to feel that it is worth the effort. The problem is compounded by the fact that programmers usually underestimate the work involved in building new software products. In addition to these problems, the cost of reuse will increase if the software or documentation is available in a less optimal form. For example, the cost of reusing software that is available only as printed listing is much higher, due to keypunch costs and errors, than software available on a standard-format magnetic tape.

2.3.2.3.2 Scope of Functions

One group of key characteristics of reusable software relate to the scope of the functions performed by the software. These characteristics are specificity of function, commonality of function, and completeness of function. Specificity of function is needed for reusability because a module that does a single well-defined function is more likely to be reused than one that performs several, perhaps unrelated, functions. This is because the inclusion of unwanted functions may make the module inefficient or hard to verify as correct, and may necessitate costly modifications to remedy this.

Specificity of function is related to Myers' (Myers 75) concept of module strength. Module strength ranges from coincidental strength, where unrelated items make up a module, to functional strength, where the entire module performs a single function. One way to determine the specificity of function of a module is to write a sentence describing its purpose:

- if the sentence is a compound sentence, has more than one verb, or contains a comma, then the module probably performs more than one function
- if the object of the verb is not a single specific item, then the module probably performs more than one function
- if the sentence contains time relationships (e.g. first, when, after, etc) then more than one function is probably performed

Commonality of function, the usefulness of the function(s) that the software performs to other applications, determines the potential for reuse of a software product. For example, a "square root" function exhibits a high commonality of function, while a special purpose simulator may not.

Finally, completeness of function is important to reusability because a module, subsystem, or system that doesn't perform a "complete" function (as defined by the user) may be costly to modify to incorporate the missing features. An example of a system performing an "incomplete" function is an editing program without a string substitution capability.

2.3.2.3.3 Independence

Another group of key characteristics relate to the software's independence on its operating environment. Specifically, these are machine independence, software system independence, and data independence.. Any independence of the software on its operating environment will make the software easily reused in a different application environment. Software dependence on a virtual operating system (Hall 80) interface, however, will not directly decrese the reusability of the software because the virtual operating system is, itself, implemented in an environment independent manner.

Several architectural features have an impact on the potential reusability of software system. These are standard features of architecture, microcode independence, and memory addressing limitations. The impact of these architectural features is explored in

the following paragraphs. Standardization of computer architecture can increase the potential reuse of software by increasing the number of environment in which the software can be executed without change. This lessens the need for software to be machine independent in order to be reused. If this standardization is coupled with a standardization of operating system interfaces, the need for software system independence is also reduced.

However, it should be noted that whenever multiple architectures are available, even if each is standardized, the costs to reuse software from another architecture is not eliminated by this standardization. For example, there is significant cost when software is moved from a standard 32 bit architecture to a standar 8 bit architecture. There is also a significant cost penalty when software is moved from a standard 8 bit architecture to a standard 32 bit architecture.

Machine independent constructs in software products, such as microcode microcode independence and machine language independent code, increase the number of environments where the software can be reused. These techniques also tend to increase the software's flexibility, which has a positive effect on reusability.

Systems that have restrictive limits on memory address size, and hence program variables for unrelated data, 'hard coding' parameters and limits, and combining independent functions. The cost to modify such software in order to reuse it in another application can be high.

2.3.2.3.4 Algorithms

The algorithms used in a software system have a direct impact on the ease of reuse of a software system. An algorithm that functions well over a wide range of inputs will generally require less modification before it can be reused. In addition, an algorithm that is accurate and efficient can be used in more software systems with few changes. Finally, the use of table-driven algorithms will, if properly designed, produce highly reusable software which can be easily adapted to different applications. For example, a table-driven parsing algorithm can be easily changed to accept a new input language; but a recursive descent parser which is not table-driven is useful only for a single language.

The availability of algorithm certification, test data, and test reports also has an impact on the reusability of software. The risk that the software will not meet later requirements will make it unlikely that anyone would choose to reuse software without these. If these items are available, however, suitability of the software for other applications can be easily assessed.

2.3.2.3.5 Languages-Acceptability

The use of nonstandard, unacceptable languages in software can increase the cost of reusing software, and may preclude the reuse of code entirely. For example, nonstandard FORTRAN extensions, if used in a software module, can make reuse on another machine difficult and costly. Also, if software is implemented in a non DOD-approved language, such as LISP, the reuse of it in a DOD product is not acceptable under normal circumstances. This may necessitate recoding in an acceptable language. However, it should be noted that coding and code level testing are typically less than 20% of development costs.

2.3.2.3.6 Fault Tolerance

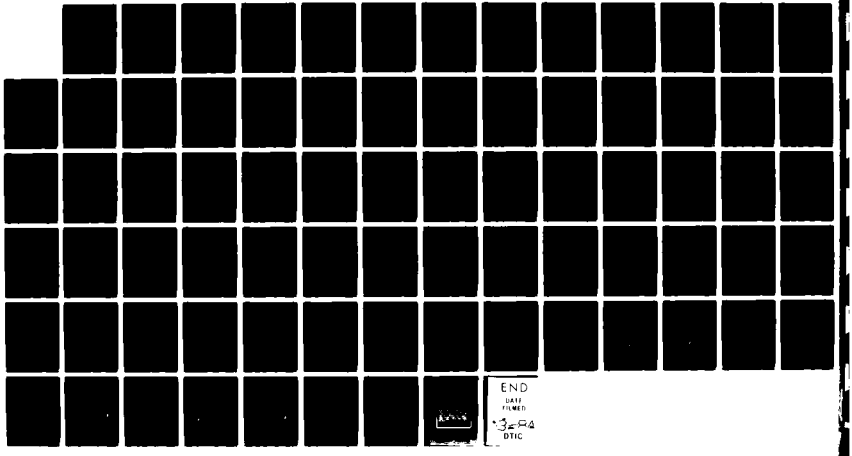
The higher the level of fault tolerance in a software product, the more reusable it will generally be. Fault tolerance in software will increase the number of environments and applications where the software can be used. This is because, in aerospace and weapons systems, a software fault can be a catastrophic event if not handled properly. For instance, a sensor out-of-range value, due perhaps to lightning, may cause an autopilot system to fail, which could result in the crashing of the airframe. Fault tolerance in the software may enable it to ignore the incorrect data and continue operation. Many military and aerospace systems require such fault tolerance. If software meets this requirement without modification, it can be reused at a low cost.

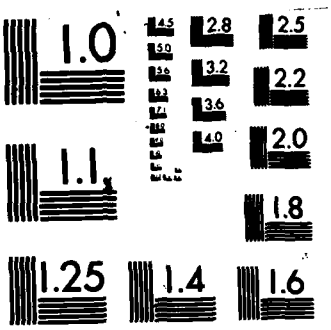
2.3.2.3.7 Documentation

The availability, accuracy, focus, style, and completeness of documentation for software systems will influence the cost to reuse them. Inaccuracies or incompleteness in specification or design documents will increase the difficulties encountered in determining the adequacy of the software for use in another application, and the cost of any

AD-A138 477 SOFTWARE INTEROPERABILITY AND REUSABILITY VOLUME 1(U) 2/2
BOEING AEROSPACE CO SEATTLE WA P E PRESSON ET AL.
JUL 83 RADC-TR-83-174-VOL-1 F30602-80-C-0265

UNCLASSIFIED F/G 9/2 NL





necessary modifications. Poorly written documentation requires a considerable effort to understand it. Worse still, the unavailability of such documentation would require these documents to be rewritten before the software could be reused, vastly increasing the costs to reuse. The following paragraphs review the impact of the availability of the documentation and its usefulness.

Available documentation comes in many forms such as requirements specifications, design specifications, development specifications, product specifications, user manuals, operating instructions and maintenance manuals. Basically these documents can be divided into two groups, User documents and Design documents.

User documents describe how to use the system. System users must have the following kinds of information:

- How to bring up the system on the computer
- How to use it
  - Preparing data
  - Interaction guide
- How to interpret its results

Design documents describe the system's structure. If modifications are to be made this kind of information is needed:

- What functions are performed
- What data is used by each function
- What is the relationship between functions
- What are the limitations of the system
- What are the performance criteria

Usefulness of documentation is the other important consideration. How usable is a document? Usefulness is a function of the information contained in the documentation and how accessible it is to the person seeking the information.

Information needs to be complete, concise and correct. Completeness is determined by reviewing the tables of contents of all system documentation to make certain the user and designer topics have been well covered. Determination of conciseness and correctness generally requires a thorough review of the contents of all system documentation.

Accessibility of information depends on documentation structure and the table of contents and index system used. A properly structured set of documents will be divided into separate volumes based on function. Each document will be divided into chapters or sections. These are subdivided into subsections and paragraphs. Each higher entity introduces the content in terms of the next lower set of entities.

Finally, the code itself should be well documented. That is, a programmer should be able to determine the function and operation of the software by reading the source code. This ensures that any changes that are required to reuse the software can be made quickly and accurately.

This hierarchical structure makes it easy to skim a document until the required information is found, then read in detail. Good documentation will have very little overlap. It is far too easy to lose consistency and correctness when information needs to be updated in more than one place.

2.3.2.3.8 Data Structures-Parameterization

The design of data structures in a software system has a great impact on the system's reusability. Generalized data structures which are easy to understand, flexible, and extensible reduce the costs associated with reusing the software. Most importantly, data structures, such as arrays, which are described parametrically are highly reusable since changes can be made in a simple, organized way.

Since changes are often required to reuse software in new applications or new environments, parametric definitions of data structures will reduce the reuse costs. However, in the case of total reuse, where no changes to the software are required, data structure design does not affect the cost to reuse the software.

2.3.2.3.9 System Complexity

The more complex a software system is, the higher the cost to reuse it will be. This is because a significant effort will be required to understand the structure and function of the system, a prerequisite to reuse.

Additionally, any modifications that are required to reuse the system will be made more difficult in complex systems, and debugging the modifications will be costly. A system with a design and function that is easy to understand will, therefore, be less costly to reuse than a more sophisticated and complex system. The areas of system complexity most relevant to reusability are module coupling and interface complexity.

The level of module coupling within a system influences the cost to reuse the software. In this instance, "module" can mean either a single procedure or a functional subsystem. A high degree of coupling between two modules means that a change in one module probably requires a modification of the other. This may be because of shared data, for example through a FORTRAN common block, or because one sets a control variable of the other, or because one module makes an unconditioned transfer to some part of the other. As the coupling of modules increase, the reusability of the modules decrease. With a high degree of coupling, the modules will need extensive modification before they can be reused, thereby increasing the cost to reuse them.

It should be noted that two modules may have a high degree of coupling without actually interacting in an operational sense. For example, two modules may share a FORTRAN COMMON block in which each module access different data items. That is, they don't technically share data. However, if one module must have its common area definition changed to add or delete data items (or expand or contract one), then the other must be changed also. Therefore, the modules don't have an interface as described below, but they are highly coupled.

The complexity of the interfaces in a software system is a strong factor in the reusability of software modules. The complexity of a construct, in terms of human perception, is determined by the amount of information that is passed across the interface, the accessibility of the information, and the structure of the information. (Yourdon 79)

In terms of software interfaces, the amount of information passed corresponds to the number of data items used or modified across the interface. As this amount of data increases, the complexity increases. The accessibility of the information depends on how many data items are passed "implicitly" across the interface, as in a FORTRAN COMMON area, as opposed to those passed directly, as the parameters of a CALL statement are. The more items that are passed implicitly, the more complex the interface becomes. In addition, the accessibility of the information is enhanced if it is presented in a standard or intuitive, rather than unexpected form. Finally, certain information structures, such as excessive nesting or the use of negative qualification (e.g. a procedure returning a "TRUE" on a failure), tend to increase interface complexity.

2.3.2.4 Summary of Software Reusability Characteristics

Section 2.3.2.3 identifies software characteristics which impact reusability. Some of these characteristics affect the cost to reuse the software, and some affect the potential of the software for reuse. Table 2.3-3 lists the characteristics described in section 2.3.2.3 and assesses how much each determines the ease, or cost, of reuse and the potential for reuse. The table demonstrates that several characteristics, such as commonality of function, primarily affect the potential of software for reuse. Other characteristics, such as data structures, mainly impact the ease of reuse of the software.

TABLE 2.3-3 Impact on Reusability by Characteristics

SYSTEM CHARACTERISTICS	IMPACT ON REUSABILITY	
	EASE OF REUSE	POTENTIAL FOR REUSE
Accessibility	M	H
Scope of Functions		
Specificity	H	H
Commonality	L	H
Completeness	M	H
Independence		
Machine Independence	H	H
Software System Independence	H	H
Data Independence	H	H
Standard Computer Architecture		
Microcode Independence	H	H
Memory Address Limitations	H	L
Algorithms	H	H
Languages-Acceptability	M	H
Fault Tolerance	M	H
Documentation		
Completeness	M	H
Organization	H	M
Data Structures-Parameterization	H	L
System Complexity		
Interface Coupling	H	L
Module Coupling	H	L

Legend:

H - High level of determination

M - Medium level of determination

L - Low level of determination

2.3.3 Reusability Criteria

This section identifies the criteria, Table 2.3-4, that are groupings of the system characteristics discussed in previous sections, which impact the reusability of software. Some of those criteria affect the cost to reuse the software, and some affect the potential of the software for reuse. These criteria are defined in Table 2.3-5.

TABLE 2.3-4 System Characteristics and Reusability Criteria

SYSTEM CHARACTERISTICS	REUSABILITY CRITERIA
Data Independence Data Structure Standard Computer Architecture Microcode Independence Algorithms	Application Independence
Documentation Completeness Organization	Document Accessibility
Scopes of Function Specificity Commonality Completeness	Functional Scope
Machine Independence Memory Addressing Limitation	Generality
Software System Independence Machine Independence	Independence
Module Coupling	Modularity
Interface Complexibility	System Clarity
Languages	Self-Descriptiveness
Languages Acceptibility	Simplicity

TABLE 2.3-5 Definition of Reusability Criteria

REUSABILITY CRITERIA	DEFINITION
APPLICATION INDEPENDENCE	Software possesses the characteristics of application independence to the extent that the software implementation is independent of database system, system libraries, microcode, computer architecture and algorithms.
DOCUMENT ACCESSIBILITY	Software possesses the characteristics of document accessibility to the extent that it provides an easy access to software documents, software source listing and selective use of the software programs components.
FUNCTIONAL SCOPE	Software possesses the characteristics of functional scope to the extent that it provides scope of functions required to be performed i.e. function specificity, function commonality and function completeness.
GENERALITY	Software possesses the characteristics of generality to the extent that it provides breadth to the functions performed.
INDEPENDENCE	Software possesses the characteristics of environment independence to the extent that the implementation of software is independent of the software operating system and machine hardware system.
MODULARITY	Software possesses the characteristics of modularity to the extent that a logical partitioning of software into independent parts, components, and modules has occurred.

TABLE 2.3-5 (Continued)

SYSTEM CLARITY	Software possesses the characteristics of system clarity to the extent that it provides a clear descriptions of program structure in the most non-complex, easy understandable and easy modifiable manner.
SELF-DESCRIPTIVENESS	Software possesses the characteristics of self-descriptiveness to the extent that it contains information regarding its objectives, assumptions, constraints, inputs, processing, outputs, components, etc.
SIMPLICITY	Software possesses the characteristics of simplicity to the extent that it lacks complexity in organization, language, and implementation techniques and is constructured in the most understandable manner.

The resulting reusability quality framework is shown in Figure 2.3-2

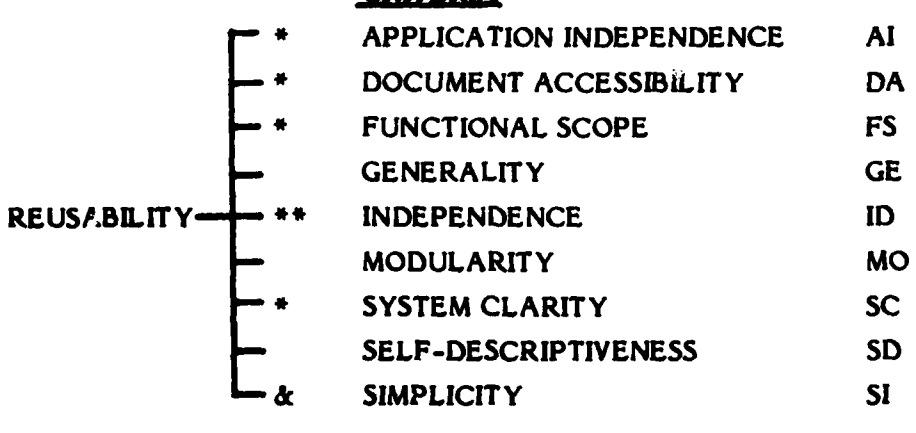


Figure 2.3-2 Reusability Framework

The following subparagraphs explain futher the detailed description of the reusability criteria.

2.3.3.1 Application Independence

The characteristics of application independence are database system independence data structure, architecture standardization, microcode independence, and algorithm. The design of data structures in a software system has a great impact of system's reusability. Generalized data structures which are easy to understand, flexible, and extensible reduce the costs associated with reusing the software. Data structures, such as arrays, which are described parametrically are highly reusable since changes can be made in a simple, organized way. Standardization of computer architecture can increase the potential reuse of software by increasing the number of applications in which the software can be implemented without change. Use of machine dependent constructs in software products, such as microcode and machine language code, will reduce the number of applications where the software can be reused. Use of microcode also tend to reduce the software's flexibility, which has an adverse effect on reusability. An algorithm that functions well over a wide range of inputs will generally require less modification before it can be reused. The use of table-driven algorithms will, if properly designed, produce highly reusable software which can be easily adapted to different applications.

2.3.3.2 Document Accessibility

Documentation is defined here as the program functional specifications and design descriptions, the user's guides, the test specifications and results, the flow charts, and the program source listings that are delivered as part of the products of software project. The availability, accuracy, focus, style, and completeness of documentation for software systems will influence the costs to reuse them. Poorly written documentation requires a considerable effort to understand it. Inaccuracies or incompleteness in specification or design documents will increase the difficulties encountered in determining the adequacy of the software for use in another application, and the cost of any necessary modifications. Accessibility of information depends on the documentation structure, and table of contents and index system used. The documentation with hierarchical structure will make it easy to skim through until the required information is found, then read in detail. To make the software more reusable then the documents should be accessible to everybody and the documentation is in public domain.

2.3.3.3 Independence

The characteristics of independence are machine independence and software operating system independence. Any dependence of the software on its operating environment can be costly to correct should the software be reused in a different environment, which often is the case. The software with those environment dependent parts isolated out from environment independent parts will tend to make it more easily modifiable and reusable. The software that uses only the simplest operating system facilities, and does not use system library routines will tend to be more reusable.

2.3.3.4 Functional Scope

The characteristics of functional scope are specificity of function, commonality of function, and completeness of function. Specificity of function is needed for reusability because a module that does a single well-defined function is more likely to be reused than one that performs several, perhaps unrelated, functions. This is because the inclusion of unwanted functions may make the module inefficient or hard to verify as correct, and may necessitate costly modifications to remedy this. *Specificity of function is related to* Myers' (Myers 75) concept of module strength. Module strength ranges from coincidental strength, where the unrelated random items make up a module, to functional strength, where the entire module performs a single integral function. Commonality of function, the usefullness of the function(s) that the software performs to other applications, determines the potential for reuse of a software product. For example, a "square root" function exhibits a high commonality of function, while a special purpose simulator may not. Completeness of function is important to reusability because a module, subsystem or system that doesn't perform a "complete" function (as defined by the user) may be costly to modify to incorporate the missing features.

2.3.3.5 Generality

The characteristic of generality implies that the software is generally designed and implemented. The machine interface functions are isolated to a few controlled modules. If the software has the machine interface functions isolated, then the software can be reused in a different machine with the same application by only modifying the machine interface parts. The characteristics of generality also imply that the I/O functions and processing functions are not mixed in a single module. Since the I/O functions are in

general, machine dependent, whether or not they are separated from processing functions will determine the difficulty to reuse the software. If the module processing is not data value and data volume limited and the module is more general and more reusable.

2.3.3.6 Modularity

Modularity is the concept of confining specific design decisions or functions into a distinct design element which is as independent of other elements as possible. This independence helps to localize the impact of modifications to within one or a few modules. The advantage of such a modularly designed system is that modules can be replaced or modified without disturbing system functions so long as their interface meets the stated requirements. The primary design goal is to produce a design of the modular structure of the program so that the modules are highly independent. The parts of a module join forces to perform a single specific well-defined function and the data should be explicitly passed as parameters or arguments. In other words we want to maximize the relationships among parts of each module (module strength) and to minimize the relationships among modules(module coupling). As more and more modular programs are implemented, libraries of reusable modules can be accumulated. These modules can be used in various places in the same program or reused as building blocks in future programs. If the modular programs are designed correctly, every module has the potential to be reused.

2.3.3.7 System Clarity

The system clarity, in both design and implementation, is one of the keys to easily understandable and easily modifiable software. System clarity is enhanced by modularity, by structured code and top-down design and implementation. The module is designed with simple control structure and in accordance with a prescribed standard. Module interfaces can be easily identified and understood. The software system in which a module interfaces with other modules via passed parameters in calling sequences, is likely to be more expandable, more extensible and require less effort to reuse in other application than one making extensive use of global or shared variables. Improvement in system clarity will tend to decrease the amount of mental effort required for comprehension of the program. This increase in comprehensibility will reduce the effort to modify and reuse the software programs. To improve the system clarity then the communication program flow, and functional application should not be complex, and the program structure should be clear and without any impurity or ambiguity.

2.3.3.8 Self-Descriptiveness

The concept of self-descriptiveness implies that the software contains enough in formation for the reader to determine or verify its objectives, assumptions, constraints, inputs, processing, outputs, components, etc. This quality is very important in being able to understand the software. The documentation contains useful explanations of software program design. The objectives, assumptions, inputs, etc. are useful at least in varying degrees of detail in source listing. The intrinsic descriptiveness of source code commentary will greatly aid efforts to understand the program operation. To make software understandable and reusable, the program source code should contain an explanations of program functions, assumptions, inputs, outputs, etc. in different detail.

2.3.3.9 Simplicity

The concept of simplicity implies that the software is lacking in complexity. The more basic techniques, structures, etc. are used, the simpler the software will tend to be. The use of a high order language as opposed to an assembly language tends to make a program simpler to understand. The quantitative counts (number of operators, operands, nested control structures, nested data structures, executable statements, statement labels, decision points, parameters, etc.) will determine to a great extent how simple or complex the source code is. Simplicity, in both design and implementation, will tend to make programs simpler, easily understandable, and easily modifiable.

2.3.4 Reusability Concepts vs. Reusability Criteria

This section examines the relationships that exist between reusability concepts (see Table 2.3-1) and reusability criteria (see Table 2.3-5). Table 2.3-6 presents the relationships between the reusability concepts and criteria; those marked with item numbers are references to the following discussions.

TABLE 2.3-6 Reusability Concepts vs. Reusability Criteria

REUSABILITY CRITERIA	REUSABILITY CONCEPTS					
	LEVEL OF REUSE			EXTENT OF REUSE		DEGREE OF REUSE
	MODULE	FUNCTION	SYSTEM	PARTIAL	TOTAL	
APPLICATION INDEPENDENCE	1	1	1	1	1	1
DOCUMENT ACCESSIBILITY	2	2	2	2	2	2
INDEPENDENCE	3	3	3	3	3	3
FUNCTIONAL SCOPE	4	4		4	4	4
GENERALITY		5	5	5	5	
MODULARITY	6	6		6		
SYSTEM CLARITY	7	7		7		
SELF-DESCRIPTIVENESS	8	8		8		
SIMPLICITY	7	7		7		

1. Application Independence is an important criterion of software reusability, and is a crucial factor in all types of reuse. Software with generalized data structures is more extensible and flexible tends to be more reusable. An algorithm that functions well over a wide range of inputs will generally require less modification before it can be reuse. Software implemented with such algorithm, although the source code sometime may not be reusable however the algorithm design can still be reused in other application.

2. Document Accessibility is a prime determinant of software reusability and cuts across all of the reusability concepts. That is, in all cases the document accessibility of software directly influences the cost to reuse it. The accessibility of requirements and design documentation determines whether the software will be considered for reuse. The accessibility of the source code and test procedures will have a large impact on the cost to reuse the software.

3. Independence, consisting of machine independence and software system independence, is a crucial factor in all types of reuse. That is, as the level of environment independence increases, the potential saving in cost to reuse the software raises with it. However, the importance of this is variable since software may be reused in the same environment. In such a case, the environment independence of software is irrelevant. In general, the software with environment independence characteristics tend to be more reusable. Although a low level of environment independence may render the software source code and test procedures unusable. The design, however, may still be reusable.

4. Functional Scope will affect the software reusability. A high level of specificity of function of system components is very important in module and functional reuse. This is because these components will be reused based primarily on the function each one performs. In system reuse, however, only the function of the entire package is important. In order for total reuse to be feasible, the scope of functions performed must be almost exactly those needed in the application. Otherwise, partial reuse would be necessary. Due to the necessity to completely understanding each function, partial reuse will usually cost more than total reuse. The degree of reuse of a software product is influenced by the specificity of function of system components. Actual

source code may be costly to reuse if it has a low level of specificity of function. However, the design and test procedure may still be cost-effective to reuse with some modifications.

5. A module is considered to be more General in nature if it is used by more than one module. If the module is generally designed and implemented, then it will takes less modification for module, function or partial reuse of the software.

6. The Modularity of software is the key factor of module, function, and partial reuse. Since this type of reuse entailes interfacing of the existing software with new software, the modularity of the existing software will determine the cost of reuse the software.

7. The Simplicity and System Clarity are the important characteristics of software comprehensibility. In the module, function or partial reuse of software, how simple and clear the software is will determine the costs to reuse the software.

8. The Self-Descriptiveness is very important in being able to understand the software. For the module, function or partial reuse of the software, there is a necessity of completely understanding the software. After the functions performed have been identified, and location for modifications are established, then the change for reusable application can be made.

2.3.5 REUSABILITY METRICS

This section identifies the metrics of software reusability. The metrics of software reusability reflect that the required software is understandable (functions performed can be easily identified, locations for modifications can be easily established, and changes for reusable applications can be easily made), modifiable (enhancements, extensions and changes can be easily made), and adaptable (can be easily altered to fit different application). There are a number of metrics that implementors of reusable software should be aware of. The metrics can be grouped as in Table 2.3-7 using the reusability criteria discussed in section 2.3.3 and 2.3.4. Not every metrics is equally important and some can be easily dealt with by every designer and implementor.

TABLE 2.3-7 Software Reusability Quality Metrics

CRITERIA	METRIC	ACRONYM
APPLICATION INDEPENDENCE	DATABASE SYSTEM INDEPENDENCE	* AI.1
	DATA STRUCTURE	* AI.2
	ARCHITECTURE STANDARDIZATION	* AI.3
	MICROCODE INDEPENDENCE	* AI.4
	ALGORITHM	* AI.5
DOCUMENT ACCESSIBILITY	ACCESS NO-CONTROL	* DA.1
	WELL-STRUCTURED DOCUMENTATION	* DA.2
	SELECTIVE USABILITY	* DA.3
INDEPENDENCE	SOFTWARE SYSTEM INDEPENDENCE	** ID.1
	MACHINE INDEPENDENCE	** ID.2
FUNCTIONAL SCOPE	FUNCTION SPECIFICITY	* FS.1
	FUNCTION COMMONALITY	* FS.2
	FUNCTION COMPLETENESS	* FS.3
GENERALITY	REFERENCE GENERALITY	GE.1
	IMPLEMENTATION GENERALITY	GE.2
MODULARITY	MODULAR IMPLEMENTATION	MO.2
SYSTEM CLARITY	INTERFACE COMPLEXITY	* SC.1
	PROGRAM FLOW COMPLEXITY	* SC.2
	APPLICATION FUNCTIONAL COMPLEXITY	* SC.3
	COMMUNICATION COMPLEXITY	* SC.4
	STRUCTURE CLARITY	* SC.5
SELF-DESCRIPTIVENESS	QUANTITY OF COMMENTS	SD.1
	EFFECTIVENESS OF COMMENTS	SD.2
	DESCRIPTIVENESS OF IMPLEMENTATION LANGUAGE	SD.3
SIMPLICITY	DESIGN STRUCTURE	& SI.1
	STRUCTURED PROGRAMMING	& SI.2
	DATA AND CONTROL FLOW COMPLEXITY	& SI.3
	CODE SIMPLICITY	& SI.4

* = New
& = Added
** = Modified

2.3.6 IMPACTS OF APPLICATIONS ON REUSABILITY

The type of application performed by a software system has a significant impact on its potential for reuse. Many software applications have specific characteristics that distinguish them from other applications. This section will examine several common applications and explore them in terms of the concepts and criteria of reusability.

2.3.6.1 Database and MIS

Most information management systems perform a comprehensive set of functions for some specific user need. Often this set of functions is well defined and needed over a long period of time with little modification. An example of such a software system is an inventory program.

In this environment, reusability usually is taken to mean the complete reuse of the entire software system. This then implies that reusability of such systems is essentially the same as portability. However, information system reuse often necessitates the reuse of the system's information as well as its software. For this to be possible, either the software system's internal data must be in a machine independent, portable form, or one of the functions included within the system must be the translation of information to and from a machine independent form. An example of this distinction of data storage in software systems is that between sequential formatted FORTRAN files, which are machine independent, and direct access unformatted FORTRAN files, which are not.

The reusability of information management systems also depends on the level of other related quality factors, including operability, correctness, and reliability. That is, the software is not likely to be reused if it doesn't also attain a reasonable level of these factors.

2.3.6.2 Missile Systems

In order for missile systems software to be reusable, it must be flexible and portable. In addition, since missile systems often perform life-critical functions, they must be highly reliable. Finally, these systems are invariably at the leading edge of technology and require very high level of efficiency to be practical. It is well established that several of

these factors are at odds with each other, since the generality needed for flexibility and portability will increase software overhead and, consequently, decrease the software's efficiency. (McCall 79-1)

This analysis shows that it is very difficult to construct reusable missile software that is still viable. However, not all parts of a missile system must have extremely high levels of efficiency. Knuth (Knuth 71) has demonstrated that, typically, 50% of the CPU time is spent executing 5% of the code. This would imply that as much as 80%-90% of the software can be implemented with sufficient portability and flexibility to be highly reusable. This would correspond to a partial functional reuse with a possibility of some total module reuse.

To achieve it, a system designer should identify those parts of the system where high efficiency is required. In these parts, only a medium level of reusability should be specified. In other parts of the system, reusability factors should take precedence over efficiency. Finally, the interface between these parts must be specified in a way which will allow the nonreusable parts to be easily replaced at a later time.

Although some parts of the system may not be reusable at code level, the design of these parts is highly reusable since the functions they perform are needed in most missile systems. This then requires that requirements and design documents be available and include the design 'as built'.

2.3.6.3 Support Software

The reusability potential of support software depends largely on the functions performed. Some support software is highly reusable because it is largely application independent, and applicable to most software development efforts. That is, the software exhibits a high degree of commonality of function. A few examples of this type of software are database management systems, floating point packages, and variable cross-reference generators. Other support software is less reusable because it is application dependent or because it conforms to a standard for some subset of the programming community. This software has a lower degree of commonality of function. An example of such a system is a special-purpose simulator, such as for a particular network architecture.

2.3.6.4 Executive Software

Executive software lies somewhere between missile software and support software in terms of efficiency constraints. Although efficiency is not generally the overriding concern in executive software, it is often important. In this application a moderate level of reusability should be specified in the system requirements in non-time-critical segments. Reuse in this type of software would most likely be partial reuse of a system, although partial and total module and functional reuse is also possible.

2.3.6.5 C3I Software

Although C3I software has historically been plagued by problems with software, it is one area where reusability has a high payoff potential. There are several well defined and well understood subsystems in C3I software that can be performed by reusable software products. These include communications, display, and man/machine interface subsystems. In such cases, the type of reuse is total functional reuse and it is very much similar to the use of mathematical libraries. Other subsystems have a lower reuse potential because they generally require high levels of security and integrity that could be compromised by reuse.

2.3.7 Reusable Software Development Guidelines

In general, it is not possible to develop software in a completely top-down manner if reusable software modules and subsystems are to be included. Systems employing previous software must use reusable modules as a baseline in much the same way conventional language constructs are used. This requires a large set of compatible routines which are easily accessed by a designer. Modules selected for reuse generally need high levels of flexibility, portability, and generality unless the particular application can be met with the software "as is" (i.e. total reuse case)

Systems which may produce reusable software as a by-product of their development should be designed with this in mind. That is, modules and subsystems that are identified as having a high potential for reuse should be developed with a high reusability level required of them. These modules should be specified early in development, and designed with flexibility, portability, and generality in mind.

Some of the questions that should be asked in determining potential reusability of modules and subsystems include:

- Is language to be used available on many machines, especially those likely to be used in this application (machine dependence)

- Is language to be used expected to be available and acceptable on machines which become available during anticipated reuse time?

- Is function to be performed expected to be required again in the future (commonality of function)?

- Is applicable, is specific hardware used in the application likely to be available for future systems, or will compatible hardware available? In addition, the following question should be asked to determine whether partial or total reusability should be emphasized:

- Is the function to be performed likely to be required unchanged in the future, or will it probably be slightly different?

The general design guidelines for reusability are presented in Table 2.3-8.

TABLE 2.3-8 General Design Guidelines For Reusability

* PROGRAMING LANGUAGES:
--- Use only DoD standard high order language.
--- Use only standard language features, do not use language extentions.
--- Read the standard language definition.
* SOFTWARE PROGRAM STRUCTURE:
--- Use structured design and modular approach concept.
--- Seperate input, output, and processing function.
--- Modules flow top to bottom.
--- Modules perform single integral function.
--- Single entry and single exit in each module.
--- Comment the software program source.
* OPERATING SYSTEM PROBLEMS:
--- Do not use the system library routines and system facilities if possible.
--- Use only the simplest operating system facilities.
--- Document (comment) the facilities used completely.
* MACHINE ARCHITECTURE:
--- Use standard computer architecture.
--- Source code is independent of word length and character size.
--- Data representation is machine independent.
* DOCUMENTATION:
--- Should be complete and manageable.
--- Structured according to level of detail and functions performed.
--- Contains algorithms used, limitations, and restrictions.
--- Describes the functions performed and relationships between functions.
--- Contains program flow charts and source listing.

Referring to Figure 2.3-3, the general problems associated with the software reusability can be addressed at the various review points in the software development process. The development guidelines to be addressed at each review are listed in the following paragraphs; they are intended to be used by designers and installers of reusable software. These guidelines will aid in developing potential reusable software programs with a wide range of generality in design and code and sufficient modifiable capability.

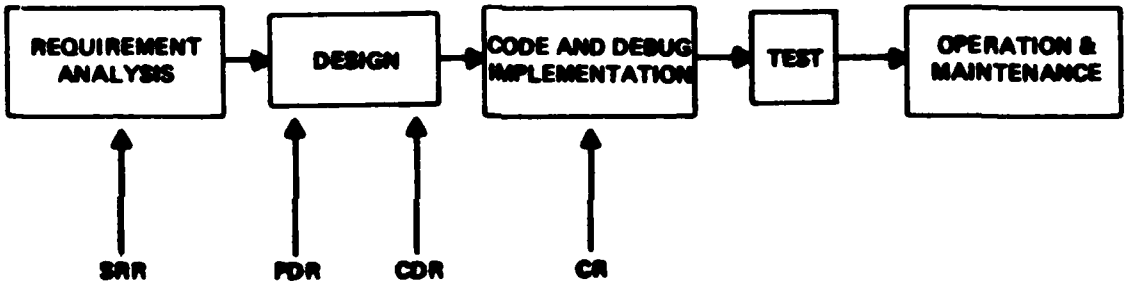


Figure 2.3-3. Software Development Phases

Minimum attention has been paid to the problem of modifying existing programs to other applications, although the decision to the problems to reuse software is usally made after the original implementation. These guidelines may still be useful to localize the problems, and determine the feasibility of software reusability.

2.3.7.1 SRR (Software Requirement Reviews)

* Plan the design and document the contents in detail to ensure a workable design that meets requirements

* Have the table of contents for all software system documents

* Determine major functions, with proper functional partitioning into subfunctions, to be performed, interrelationships between functions/subfunctions, system limitations and performance criteria

* Briefly draw software system function charts identifying and defining interface requirements at inter and intra levels

* Develop abstract models for describing and structuring modules

* Prepare statements of requirements for input data, error tolerance, and processing failure recovery

* Have information regarding how to bring up the system, prepare data, use (interactive with) the system and interpret results

* Identify the possible potential reusable applications

2.3.7.2 PDR (Preliminary Design Reviews)

* Design the software in a manner that makes it easy to reuse

* Design the software system in a top-down fashion

* Provide a modular hierarchy of the software system and identify all modules and database interfaces definition in the system

* Determine the functions to be performed and the data to be input or output by each module

* Allocate the functions and subfunctions to modules in a way that enhances modularity and functional independence

* Have each module perform a single integral function

* apply structured design technique to draw the system structure charts and show in detail the logical structure of the program

* Describe the structure of the system, explain the functions and the interfaces

* Prepare the plans to bring up the software system and test the software

2.3.7.3 CDR (Critical Design Reviews)

* Follow the hierarchical structured development concepts to design simple independent modules that have low coupling, high cohesiveness and high functional binding (i.e. maximize the intramodule strength and minimize the intermodule coupling):

  a. Top-down design (program flow always forward)
  b. Break the system into small independent modules
  c. Modules limited in size
  d. Single entry and exit in each module

e. Structured programming, avoid generating complicated hard to understand "spaghette" code

f. Temperary storage should not be shared between modules

* Construct the function (module) which facilitates or encourages its use elsewhere either in part or in total

* Construct modules which are functionally cohesive and perform single integral function

* Use modular approach to design each module with single integral function performed

* Identify the limitations of processing performance in each modules

* Define all module interfaces in a simple and explicit manner and define control data to be passed between the interfaces

* The module should be internally structured so the paran eters which customize it for each occasion are isolated and placed in a single table. The table provides the control parameters which properly route processing

* Centralize the I/O functions and separate them from computation functions

* Describe the control flow, data flow and algorithmic considerations within the various modules

* Draw the structured system flow charts, function flow charts and module flow charts

* Prepare the procedures to bring up the software system and the test procedures

* Have the table of contents, system and functional design specification and/or index for all software system documents

* Isolate machine interfaces by assigning machine interface functions to a few controlled routines. Use global data structures to define peripheral characteristics, i.e. use 'virtual' instead of physical peripherals.

2.3.7.4 CR (Code Reviews)

* Implement on standard computer architecture and use DoD (Department of Defense) standard HOL (High Order Language)

* Avoid using software system utility programs and library routines

* Each module contains a commentary header block which describes the following:

  a. Functional description
  b. Input/output parameter, local and global data items descriptions
  c. Restrictions and limitations
  d. Calling what modules(routines) and called by what modules(routines)
  e. Methods used and/or algorithm description
  f. Assumptions
  g. Error recovery types and procedures for all error exits

* Comment all machine dependent codes and all non-standard HOL (High Order Language) statements

* Comment all software system dependent utilities and routines used

* Comment all transfer of control and destinations

* Clearly specify any inter-module communication by comments

* Comment and define all parameter ranges and their default conditions

* Describe the physical or functional property represented by variable names and explained in the comments

* Limit size of modules

* Avoid using embedded constants/literals and self-modifying codes

* Make all code maximally machine-independent, logically blocked and indented

* Reduce interface communication complexity by passing parameters directly between modules if possible.

2.3.8 Tradeoff Between Reusability and other Quality Factors

The software characteristics that make software reusable have a positive effect on some of the other quality factors but a negative effect on others. The following sections describe the reasons for conflict in those with possible negative impacts, and discuss the effects of other factors on software reusability. Table 2.3-9 shows the relationship between reusability and other quality factors.

TABLE 2.3-9 Tradeoff of Reusability With Other Quality Factor

OTHER FACTORS	REUSABILITY
CORRECTNESS	+
EFFICIENCY	-
FLEXIBILITY	+
INTEGRITY	-
INTEROPERABILITY	+
MAINTAINABILITY	+
PORTABILITY	+
RELIABILITY	+ or -
TESTABILITY	+
USABILITY	+

Legend: + POSITIVE RELATION
- INVERSE RELATION

The Table 2.3-10 shows the criteria used to determine the software reusability and other quality factors which have these same criteria in common. This Table clearly shows that in order to achieve software reusability by improving certain criteria which other quality factors will also benefit.

TABLE 2.3-10 Reusability Criteria and Other Quality Factor

REUSABILITY CRITERIA	OTHER QUALITY FACTORS BENEFITTED
GENERALITY	Flexibility
MODULARITY	Flexibility Interoperability Maintainability Portability Testability
INDEPENDENCE	Portability
SELF-DESCRIPTIVENESS	Flexibility Maintainability Portability Testability
SIMPLICITY	Maintainability Reliability Testability

2.3.8.1 Correctness vs. Reusability

* Implementing software satisfy program specifications to make it easier to reuse in other application will increase correctness

2.3.8.2 Efficiency vs. Reusability

* Software generality and environment independence required to make software reusable will decrease the efficiency of the software system

* Using moduality and well-commented high level code to increase the reusability will result in less efficient operation

* The overhead required to provide reusability can often decreases the efficiency of the software system

* Use direct code or optimized system software or utilities to improve the efficiency tends to make software less reusable

2.3.8.3 Flexibility vs. Reusability

* Both reusability and flexibility are improved by the increased modularity, self-descriptiveness and generality

* Reusable software in general modifiable and flexible

* Software flexibility will be enhanced by improved reusability

* Software with flexibility will be modifiable, then it tends to be more reusable

2.3.8.4 Integrity vs. Reusability

* Generality in code, data structures and documentation required by reusable software will have a negative impact on integrity

* Additional code and processing required to control the access of the software to achieve the integrity tends to make the software less reusable

* Accessibility of system requirements, design documentation, code, and test procedures to improve reusability will adversely influences of software integrity

2.3.8.5 Interoperability vs. Reusability

* Both reusability and interoperability are improved by increasing modularity

* Separating I/O functions from other functions to achieve reusability will also improve interoperability

2.3.8.6 Maintainability vs. Reusability

* Both maintainability and reusability are improved by the increased modularity, self-descriptiveness and simplicity

* Reusable software will be easy to modify and easy to maintain

* Maintainable software, in general, is simiplicity in design and coding and fully modularized without increasing system complexity, therefore it tends to be more reusable

* Software maintainability will be enhanced by improved reusability

2.3.8.7 Portability vs. Reusability

* Both reusability and portability are improved with increasing modularity, software independence, machine independence and self-descriptiveness

* Portable software programs are considered as total reusable programs

* Software portability will be enhanced by improved reusability

2.3.8.8 Reliability vs. Reusability

* If reusability is attained through methods that guarantee proper action over a wide range of inputs, the software becomes more reliable

* If reusability is attained through a great deal of flexibility and the software is modified for each application then reliability may decrease

* If reusability is attained through simplicity of software design and coding then reliability may increase

* Reliable efficient code seems harder to reuse and reusable code introduces processing and storage overhead which decreases reliability

2.3.8.9 Testability vs. Reusability

* Both reusability and testability are improved with increasing modularity, self-descriptiveness and simpilicity

* Reusable software ought to be testable. A software program that does not perform its intend function tends not to be reusable, testability is required to prove software performs its intend function

* Software testability will be enhanced by improved reusability

2.3.8.10 Usability vs. Reusability

* Software developed for usability, i.e., to minimize the effort required to learn, operate, prepare input, and interpret output, also tends to be more reusable

* Software usability will be enhanced by improving reusability

2.3.9 Data Collection

To measure the software reusability, the metric worksheet for the software reusability have been throughly developed. Worksheet data was collected from four major projects. Some of the modules have been actually reused, some of the modules, designed with flexibility in mind, all code is maximumally machine independent, and using top-down modular design techniques, which are potentially reusable. The reusability of software is mainly concerned with module reusability, so the module worksheet data of the projects are collected. The number of modules and lines of code of the projects which the worksheet data have been collected is showed in Table 2.3-11.

TABLE 2.3-11 Worksheet Data Collected

PROJECT	1	2	3	4
NUMBER OF MODULES	28	32	20	23
LINES OF CODE	12720	3265	8589	7748

Those four projects which the worksheet data were collected have different characteristics of software reusability. One of the projects has employed the modern programming practices, one of the projects is avionics support software which have been reused in different applications, one of the projects is developed into two phases due the requirements been changed, however the phase 1 software had been reused in the phase 2 development, and finally one of the projects, the software have been reused in the contract development due to using different kind of computers.

A software program was implemented to compute the reusability metrics from the reusability raw worksheet data. The reusability module metrics are then obtained. The unweighted average system total metric score for each criteria, and the system total metric score for the factor were computed for each project. The results are shown in Table 2.3-12, where AI stands for Application Independence, DA for Documentation Accessibility, FS for Functional Scope, GE for Generality, ID for Independence, MO for Modularity, SC for System Clarity, SD for Self-Descriptiveness, and SI for Simplicity.

TABLE 2.3-12 Metrics Summary (By Criterion)

REUSABILITY CRITERIA	PROJECT			
	1	2	3	4
AI	.90	.74	.73	.74
DA	.85	.90	.90	.90
FS	.96	.52	.48	.50
GE	.63	.60	.68	.39
ID	1.00	.63	.74	1.00
MO	.85	.71	.66	.67
SC	.81	.64	.54	.58
SD	.76	.42	.34	.41
SI	.68	.64	.40	.37
PROJECT AVERAGE	.83	.64	.61	.62

The reusability productivity, i.e. resulting lines of code developed during conversion of program per mam-month, is also collected for each project. Note the conversion efforts include the software requirement analysis, program design and implementation (coding). In Table 2.3-13, the productivity is presented with their corresponding reusability metric score, and Figure 2.3-4 shows the plot of the productivity versus the reusability factor metric.

TABLE 2.3-13 Productivity vs. Reusability Metrics

PROJECT	1	2	3	4
PRODUCTIVITY (L.O.C./M.M.)	79	24	12	15
REUSABILITY METRIC SCORE	.83	.64	.61	.62

2.3.9.1 Validation of Reusability Metrics

The reusability rating value is defined as

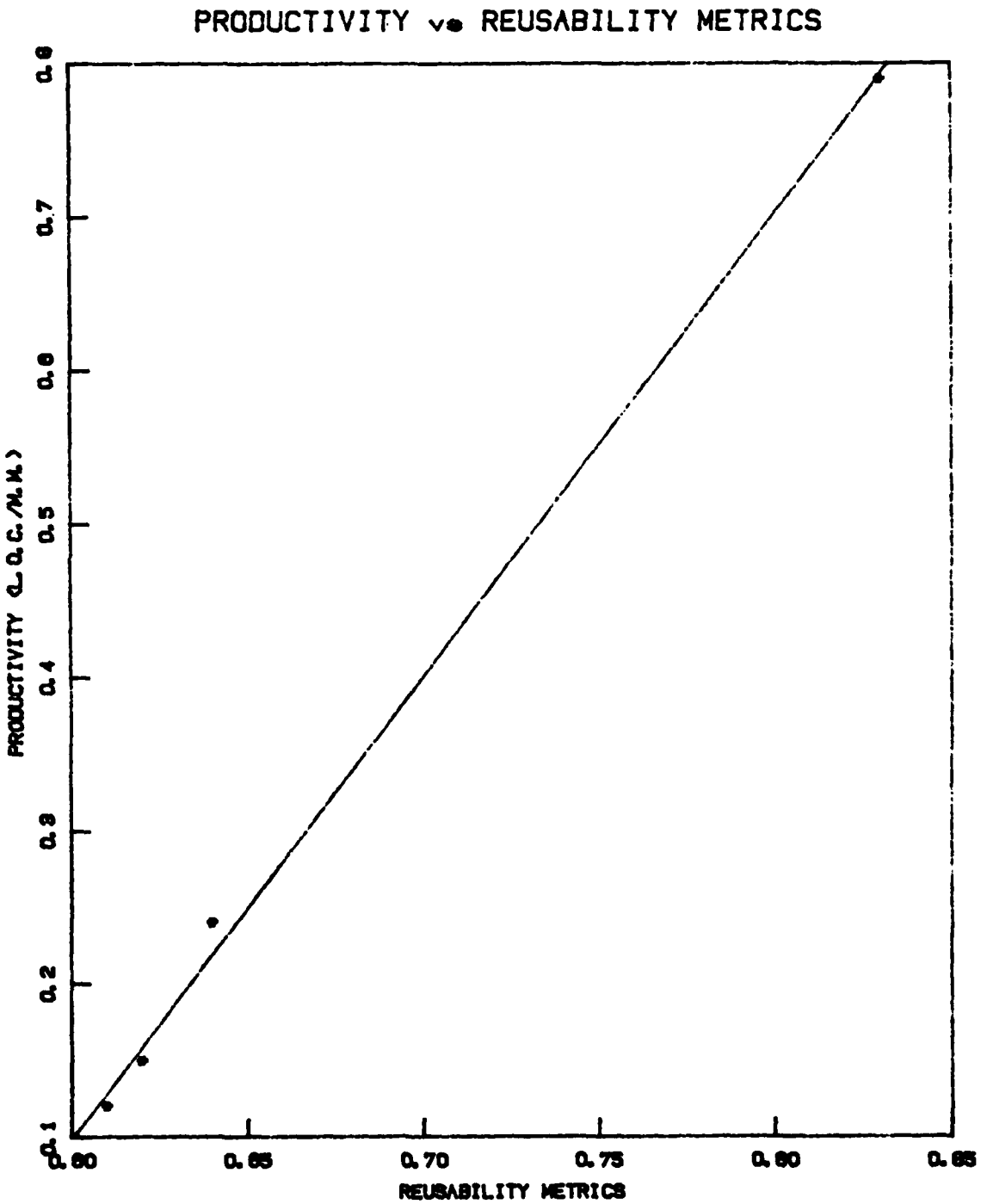


Figure 2.3-4: Productivity vs. Reusability Metrics

$$R = 1 - C / D$$

where "C" is the effort to convert a program in a reusable application, and "D" is the effort to initially develop the program. During the validation of reusability metrics some difficulty in obtaining the exact rating value for reusable application was experienced. First, information related to reusable conversion efforts on any particular module was not available. Second, the efforts spent on improving program algorithms, logics, operations, and etc. could not be separated from pure reusable conversions. However, the productivity figures are available for each project, which show that the higher the reusable metrics score the higher the reusability productivity. In other words, the higher the reusable metrics score the easier to reuse the software; it takes less time to convert the program for other application. That is

$$C \Rightarrow K / M$$

where " M " is the average total metric score for reusability, and "K" is a constant Then,

$$R \Rightarrow 1 - (K / M) / D = 1 - K' / M$$

where "K'" is equal to "K/D" and is a constant for a particular project. And,

$$K' \Rightarrow (1 - R) * M$$

For any particular project, the data on conversion efforts and development efforts provide the capability to compute the system rating value for reusability. Knowing the system rating value and the average total reusable metric score, the constant "K'" can then be computed. The module rating value can then be computed in the following way:

$$R \text{ (module)} \Rightarrow 1 - K'/ M$$

where "M" is average module metric score. The increase of productivity due to the employment of modern programming practices is also considered in the computation of "K'". In Table 2.3-14, the module reusability rating values are presented.

TABLE 2.3-14 Summary of Module Reusability Rating Values

PROJECT-1 MODULE REUSABILITY RATING VALUES
.24 .33 .28 .32 .32 .29 .31 .31 .34 .36 .37 .35 .35 .35 .34 .34 .34 .32 .37 .37 .29 .33 .31 .35 .32 .34 .32 .32
PROJECT-2 MODULE REUSABILITY RATING VALUES
.34 .35 .29 .28 .26 .24 .29 .28 .35 .31 .28 .30 .28 .35 .27 .30 .29 .29 .35 .30 .32 .31 .29 .29 .26 .26 .26 .29 .30 .27 .28 .31
PROJECT-3 MODULE REUSABILITY RATING VALUES
.21 .19 .28 .25 .20 .22 .20 .26 .17 .24 .31 .29 .19 .18 .28 .16 .17 .28 .28 .27
PROJECT-4 MODULE REUSABILITY RATING VALUES
.25 .26 .24 .30 .25 .24 .30 .23 .23 .24 .24 .25 .26 .26 .28 .26 .26 .25 .25 .22 .23 .24 .28

Based on the above arguments, the least square linear regression on the module rating value and module metrics is then performed. Routines to perform the analysis were developed on the VAX 11/780 and plotting is done using "S" and the HP7221 plotter. Since the metric and quality factor rating were normalized positive values, all data points fall within the positive quadrant of the graph. We assume:

$$Y = A + B * X$$

where "Y" represents the rating value, "X" is the metric score, "A" is the Y-intercept value, and "B" is the slope. For each set of metric data ("X") the corresponding slope coefficient ("B") and Y-intercept ("A") are then determined by the least square linear regression. That is

$$B = (N * \sum Xi\ Yi - \sum Xi * \sum Yi) / (N * \sum Xi\ Xi - (\sum Xi)*(\sum Xi))$$
$$A = Mean(Y) - B * Mean(X)$$

where the $\sum$ is summed from i=1 to i=N and N is the number of samples, Mean(Y) is the sample average of Y data, and Mean(X) is the sample average of X data. The least square

linear regression analysis program has been implemented. The standard deviation of module metric and rating values, slope coefficient("B"), Y-intercept("A"), standard error of estimate, correlation coefficient of the linear regression analysis results of Project-1 Metric SC.1 are presented in the Table 2.3-15. In Appendix-A, all the least square linear regression results are presented. In Appendix-C, the rating and metrics data are plotted with least square linear fit lines, the 90 percent confidence interval for the data is also illustrated with dashed lines. In Appendix-B, the least square linear regression results on all four projects combined data are presented.

TABLE 2.3-15 Regression Analysis Example

PROJECT-1	METRIC SC.1	REUSABILITY RATING
Average	.72	.32
Range	.60-.81	.23-.37
Standard Deviation	.059	.029
Slope Coeff.("B")	.40	
Y-intercept("A")	.041	
Std. Error of Estimate	.018	
Corr. Coeff.	.80	

2.3.9.2 Observations on Data Collection Results

The results obtained for the software reusability metrics are futher analyzed by regression analysis. The result in Table 2.3-16 shows those metrics that have correlation with reusability. Those metrics without any correlation due to no variation in the metric data, do not show in the Table.

TABLE 2.3-16 Reusability Metrics With Correlation

METRIC	PROJECT			
	1	2	3	4
AI.3			Y	
FS.1	Y	Y	Y	Y
GE.2	Y		Y	
ID.1			Y	
ID.2			Y	
MO.2	Y	Y	Y	Y
SC.1	Y	Y	Y	Y
SC.2	Y	Y	Y	Y
SC.4	Y	Y	Y	Y
SD.1	Y	Y	Y	Y
SD.2		Y	Y	Y
SD.3	Y	Y	Y	Y
SI.1	Y		Y	Y
SI.3	Y	Y	Y	Y
SI.4	Y	Y	Y	Y

Y: Good Correlation

In evaluating reusability metrics, a number of techniques were utilized, to provide a desirable balance between inituitive judgment and analytical objectivity. Plots were prepared of all data collected for each metric, both individually by project, and collectively, by pooling the project data. When there was sufficient data, regression lines

for each metric were constructed for the projects individually, as well as pooled. Pooling data sometimes increased the correlation for metrics where project data was sparse and/or highly scattered. From comparison of graphic presentations of the data, including the regression lines and the associated confidence bounds, it was possible to classify certain metrics as too project sensitive, at one extreme, and too insensitive to variation in metric level, at the other.

An alternative screening approach was also exercised, involving completely objective statistical analysis. This method involves comparison of the individual project regression lines for a metric by means of simple analysis of covariance. The details of this technique can be found in section 11.14 of Statistical Theory and Methodology in Science and Engineering, Second Edition by K.A. Brownlee (John Wiley & Sons, Inc., New York, NY, 1965).

Essentially, the method tests the hypothesis that the differences between regressions obtained either from several individual groups of data or by merging the groups are attributable to chance alone.

For the reusability metrics, tests were performed at the 2.5% level of significance, meaning that the hypothesis under test would be falsely rejected with probability not greater than 0.025. The test was applied to nine metrics regressed with data from four projects and two each regressed with data from three and two projects, respectively. The results generally confirmed the findings of the more intuitive graphic approach. However, in two instances the method accepted the hypothesis under highly counterintuitive conditions, indicating that its objectivity should not be regarded as an infallible substitute for qualified judgment.

In Appendix B, plots are presented of the regressions for the metrics selected by the combined screening techniques mentioned above. Those pooled on the basis of the analysis of covariance alone are FS.1, GE.2, ID.2, MO.2, SC.4, SD.1, SI.1, and SI.3. Additional metrics pooled on the basis of visual comparisons of project plots, pooled plots, and plots supplemented with project data for which regressions could not be made but which correlated well with other pooled data are SC.1, SC.2, SD.3 and SI.4.

In Appendix D, a multivariate (multiple regression) analysis is presented. Those metrics that have correlation with reusability and those that are project insensitive, are included in the multiple regression equation. This amounts to eleven independent variables (metrics). The details of this analysis can be found in sections 11.1-11.3 of Applied Linear Statistical Models, by J. Neter and W. Wasserman (Richard D. Irwin, Inc., Homewood, Illinois, 1974).

Simply, the technique analyzes the intercorrelations between the metrics by developing a correlation matrix. The presence of many intercorrelations among the metrics provides a rationale to screen the metrics to obtain a representative subset. The metrics that are highly correlated with reusability are also highly correlated amongst each other. This characteristic may add little to the predictive power of the regression equation. Therefore, the next step is to find a subset of metrics. The method used is the all possible regressions search procedure. This involves examining all regression equations containing the eleven independent variables (metrics) and the selection of the subset of variables based on some criterion. The R^2 and Cp criteria, of the above mentioned text, were used to develop Table D1 in Appendix D.

2.3.10 Conclusions

Software reusability has been measured on four different projects both at the system level and module level. From studying the trend of software reusability in four different projects, the conclusion has been reached that the higher the reuseability metrics scores, the less effort needed to convert the program in reusable application, in other words, the higher the reusability rating.

From the module level data analysis, the metrics Ai.3, FS.1, GE.2, ID.2, MO.2, SC.1, SC.2, SC.4, SD.1, SD.2, SD.3, SI.1, SI.3, and SI.4 are validated and accepted in the reusability framework. Other metrics do not show significant variation in different modules and the linear regression analysis are not performed. Although the reusability quality measures could be misleading without a large reusability database, based the data available the results are very promising.

2.3.11 Recommendations

The ultimate reusability quality level is determined by the software development process itself. The probability of success of achieving reusability at acceptance (application)

time is a function of the method used by the development organization augmented by reusability quality criteria satification checklists in the development process.

Reusability of software is a measure of software independence, understandability, modifiability, and adaptability. In other words, reusable software should be environment independence, that is software system independence, machine independence, and database system independence; reusable software should be understandable, that is functions performed can be easily identified, locations of modifications can be easily established, and changes for reusable applications can be easily made; reusable software should be modifiable, that is enhancements, extensions and changes can be easily made; and reusable software should be adaptable, that is can be easily altered to fit different applications. To achieve the goal of software reusability, it should be follow the reusable software development guides, start with modular design, and implement in simple and general fashions.

Building programs from reusable software modules can significantly reduce costs. There is currently no way of knowing exactly how much DoD spends on 'support' and 'applications' software which had previously been produced for some other program or programs. The first effort to control costs in duplication of software development should begin with reusing software. With clear software development standards of software reusability for defense systems acquisition, the software development costs resulting from duplication of software can then be reduced.

Software reusability has been proved to be cost effective. To develop a reusable software may cost more initially, however in the long run not only reusability benifit but also maintainability and portability benefit. Reusability metrics have proven useful in actual practice. As shown in Figure 2.3-4 the higher the reusable metrics score the more reusable the software is, in turn the reusable productivity is higher. To develop the reusable software, reusable metrics score can be used as feedback to implementors during program development to indicate the reusability of their software product. When the metric scores for their software modules below the acceptable threshold value, implementors are instructed to consider ways to improve the software reusability. Software reusability has been proved to be cost effective. To reuse the "old" source code could save 20-25% of the production cost.

To achive the software reusability, first, the reusable software programs should be produced. Second, these reusable software programs should be accessed by the potential software implementors that is the reusable software library should be established. In this library there will have the data base contain a directory of content, functional descriptions of content categories and reusability architectures, and detailed descriptions of the reusable modules and algorithms -- including performance requirements, design documentation, source code and data, verification documentation, timing and sizing data. The remote, interactive user interface would enable access by projects implementors. The library would be generated and maintained by a particular reusable software control group. This group would accept information in the form of code, data, and documentation from outside groups and organizations and would select and/or modify incoming information, certify and release information to the library, maintain the library contents, and provide training on use of the library. Then the potential benefits of reusable library to a software project will be realized. That is productivity is increased, the development time is reduced, the risk is reduced, and the reliability is increased.

2.4 IMPACT ON AMT

The worksheets of quality metrics and metrics tables have been updated. Original metrics tables do not always give the metrics score between 0 and 1. The corrected algorithms have been included in the new metrics table. For example, to name a few: SI.1(5) now is corrected as 0.5*(1/# entrances + 1/# exists) so that SI.1(5) will never be greater than 1, MO.2(2) now is corrected as = 1 if module size is less than 100 LOC (lines of code) or = 100/LOC if module size is greater than 100 LOC. and, GE.2(2) now is corrected as 1/(1 + # machine dependent functions) so that GE.2(2) can never have an infinitive answer. The worksheets statements are also corrected as closer corresponding to metrics tables. In the new metrics tables and the metrics worksheets, some are corrections of the original errors and some are newly added to reflect the new interoperability and reusability quality metrics. This will have some significant impact on the Automated Measurement Tool. In other words, the Automated Measurement Tool should be updated and the metrics computation algorithms used in the Automated Mesurement Tool should be corrected.

2.5 QUALITY MEASUREMENT MANUAL ENHANCEMENT

The software quality measurement manual (Vol. II Software Quality Measurement Manual) presents a complete set of procedures and guidelines for introducing and utilizing current software quality measurement techniques in a quality assurance program associated with large scale software system developments. The new software quality measurement manual is more complete and self-sufficient. It contains not only the worksheets but also the metric tables. The new software quality measurement manual is now made more usable and more expandable; i.e., the following new features are added and improved: - Worksheet and metrics tables now included as appendices, easily updated, easily used - Worksheet reorganized is more parallel in structure to software development phases - Metrics tables now ordered alphabetically by criteria - Metrics tables now have corrected metrics computation algorithms - Inconsistencies have been removed (e.g. cross-reference between metrics tables and worksheets) - Worksheets more closely correspond with metrics tables.

REFERENCES

McCall 79-1 McCall, J., Matsumoto, M., "Software Quality Measurement Manual", RADC, Sept 1979.

Bowen 81 Bowen, T., "Conceptual Framework for Reusable Software", Boeing Document D180-25964-1, 1981

Hall 80 Hall, D., et. al., "A Virtual Operating System", CACM, Vol 23, No. 9, pp. 495-502, Sept 1980

Yourdon 79 Yourdon, E., Constantine, L., Structure Design, Prentice-Hall, Englewood Cliffs, N.J., 1979

Knuth 71 Knuth, D., "Empirical Study of FORTRAN programs", Software Practice and Experience, Vol. 1, No. 2, pp 105-133, April-June 1971

McCall 79-2 McCall, J., Matsumoto, M., "Software Quality Metrics Enhancements Final Report, RADC" Sept 1979

Myers 75 Myers, G., Reliable Software Through Composite Design, Mason/Charter 1975

REFERENCES FOR STATISTICAL METHODS

* Lindstone, Harold A. and Murray Turoff (Editors), "The Delphi Method - Techniques and Applications", Addison-Wesley, Reading, Mass., 1975

* Saul. I. Guss, "Linear Programming", McGraw Hill NY, 1975,

* K. A. Brownlee, "Statistical Theory and Methodology in Science and Engineering", Second Edition, John Wiley & Sons, Inc., New York, NY 1965

* J. Neter and W. Wasserman, "Applied Linear Statistical Models", Richard D. Irwin, Inc., Homewood, Illinois, 1974,

APPENDIX A

Project Regression Analysis Summary

TABLE A-1: Project-1 Regression Analysis Summary

STATISTIC	METRIC						RATING
	FS.1	GE.2	ID.2	MO.2	SC.1	SC.2	
AVERAGE RANGE	.91 .50-1.00	.95 .40-1.00	1.00 .98-1.00	.69 .26-.88	.72 .59-.81	.80 .59-.92	.33 .24-.37
STANDARD DEVIATION	.18	.14	.00	.19	.06	.11	.03
SLOPE COEFFICIENT	.13	.14	6.19	.09	.39	.22	
Y-INTERCEPT	.21	.19	-5.85	.26	.04	.15	
STANDARD ERROR OF ESTIMATE	.02	.02	.02	.02	.02	.02	
CORRELATION COEFFICIENT	.76	.70	.73	.63	.80	.82	

TABLE A-1: Project-1 Regression Analysis Summary(Continued)

STATISTIC	METRIC						RATING
	SC.4	SD.1	SD.3	SI.1	SI.3	SI.4	
AVERAGE RANGE	.71 .44-1.00	.58 .37-75	.86 .83-.95	.60 .50-62	.31 .00-1.00	.80 .74-.87	.33 .24-.37
STANDARD DEVIATION	.17	.08	.03	.04	.32	.04	.03
SLOPE COEFFICIENT	.12	.16	.29	.19	.07	.39	
Y-INTERCEPT	.24	.24	.08	.21	.31	.02	
STANDARD ERROR OF ESTIMATE	.02	.03	.03	.03	.02	.03	
CORRELATION COEFFICIENT	.74	.44	.26	.30	.72	.49	

TABLE A-2: Project-2 Regression Analysis Summary

STATISTIC	METRIC						RATING
	FS.1	MO.2	SC.1	SC.2	SC.4	SD.1	
AVERAGE RANGE	.38 .06-.50	.42 .33-.43	.41 .33-.79	.38 .07-.75	.62 .40-.87	.26 .04-1.00	.29 .24-.35
STANDARD DEVIATION	.18	.02	.08	.21	.06	.17	.03
SLOPE COEFFICIENT	.10	.75	.25	.1	.012	.07	
Y-INTERCEPT	.26	-.02	.19	.25	.29	.28	
STANDARD ERROR OF ESTIMATE	.02	.03	.02	.01	.03	.03	
CORRELATION COEFFICIENT	.63	.51	.68	.87	.03	.39	

TABLE A-2 Project-2 Regression Analysis Summary (Continued)

STATISTIC	METRIC				RATING
	SD.2	SD.3	SI.3	SI.4	
AVERAGE RANGE	.17 .17-.25	.83 .83-.84	.29 .01-1.00	.77 .67-.84	.29 .24-.35
STANDARD DEVIATION	.02	.00	.30	.04	.03
SLOPE COEFFICIENT	.96	.85	.09	.36	
Y-INTERCEPT	.13	-.42	.27	.02	
STANDARD ERROR OF ESTIMATE	.02	.03	.01	.03	
CORRELATION COEFFICIENT	.72	.08	.90	.42	

TABLE A-3 Project-3 Regression Analysis Summary

STATISTIC	METRIC						RATING
	FS.1	GE.2	ID.1	ID.2	AI.3	MO.2	
AVERAGE RANGE	.29 .10-.50	.76 .60-.80	.85 .75-1.00	.62 .25-1.00	.97 .69-1.00	.32 .20-.48	.23 .17-.31
STANDARD DEVIATION	.17	.07	.13	.36	.08	.07	.05
SLOPE COEFFICIENT	.19	.04	.32	.12	.19	.36	
Y-INTERCEPT	.18	.20	-.04	.16	.04	.11	
STANDARD ERROR OF ESTIMATE	.04	.05	.02	.02	.05	.04	
CORRELATION COEFFICIENT	.66	.06	.86	.89	.35	.53	

TABLE A-3 Project-3 Regression Analysis Summary (Continued)

STATISTIC	METRIC						RATING
	SC.1	SC.2	SC.4	SD.1	SD.2	SD.3	
AVERAGE	.37	.25	.48	.23	.16	.62	.23
RANGE	.32-.43	.04-.60	.43-.56	.01-.48	.12-.20	.51-.67	.17-.31
STANDARD DEVIATION	.03	.13	.03	.14	.02	.04	.05
SLOPE COEFFICIENT	.41	.16	.64	.28	1.81	.86	
Y-INTERCEPT	.08	.19	-.08	.17	-.05	-.30	
STANDARD ERROR OF ESTIMATE	.05	.04	.04	.03	.03	.03	
CORRELATION COEFFICIENT	.22	.43	.47	.79	.78	.82	

TABLE A-3 Project-3 Regression Analysis Summary (Continued)

STATISTIC	METRIC			RATING
	SI.1	SI.3	SI.4	
AVERAGE	.40	.05	.74	.32
RANGE	.27-.50	.00-.25	.68-.83	.17-.31
STANDARD DEVIATION	.09	.06	.04	.05
SLOPE COEFFICIENT	.43	.43	.03	
Y-INTERCEPT	.06	.21	.19	
STANDARD ERROR OF ESTIMATE	.03	.04	.05	
CORRELATION COEFFICIENT	.82	.58	.04	

TABLE A-4 Project-4 Regression Analysis Summary

STATISTIC	METRIC						RATING
	FS.1	MO.2	SC.1	SC.2	SC.4	SD.1	
AVERAGE	.33	.34	.38	.25	.48	.23	.25
RANGE	.10-.50	.26-.40	.36-.52	.02-.61	.44-.55	.03-.60	.22-.28
STANDARD DEVIATION	.15	.05	.03	.14	.03	.14	.02
SLOPE COEFFICIENT	.06	.34	.49	.11	.39	.10	
Y-INTERCEPT	.23	.14	.07	.22	.07	.23	
STANDARD ERROR OF ESTIMATE	.02	.01	.01	.01	.02	.02	
CORRELATION COEFFICIENT	.44	.76	.76	.79	.58	.67	

TABLE A-4 Project-4 Regression Analysis Summary (Continued)

STATISTIC	METRIC					RATING
	SD.2	SD.3	SI.1	SI.3	SI.4	
AVERAGE	.34	.67	.37	.05	.70	.25
RANGE	.29-.48	.66-.68	.33-.50	.00-.20	.64-.77	.22-.28
STANDARD DEVIATION	.04	.01	.05	.06	.04	.02
SLOPE COEFFICIENT	-.17	.60	.23	.35	.49	
Y-INTERCEPT	.31	-.15	.16	.24	-.09	
STANDARD ERROR OF ESTIMATE	.02	.02	.02	.01	.01	
CORRELATION COEFFICIENT	-.33	.14	.56	.92	.84	

APPENDIX B

Regression Analysis - Combined Projects

TABLE B-1: Projects-Combined Regression Analysis Summary

STATISTIC	METRIC						RATING
	FS.1	GE.2	ID.2	MO.2	SC.1	SC.2	
AVERAGE	.50	.87	.84	.46	.48	.44	.28
RANGE	.10-1.00	.40-1.00	.25-1.00	.20-.879	.33-.76	.02-.92	.16-.37
STANDARD DEVIATION	.31	.15	.29	.18	.16	.27	.04
SLOPE COEFFICIENT	.12	.27	.17	.19	.21	.14	
Y-INTERCEPT	.22	.05	.14	.20	.18	.22	
STANDARD ERROR OF ESTIMATE	.03	.04	.03	.03	.03	.03	
CORRELATION COEFFICIENT	.75	.67	.84	.73	.72	.80	

TABLE B-1: Projects-Combined Regression Analysis Summary (Continued)

STATISTIC	METRIC						RATING
	SC.4	SD.1	SD.3	SI.1	SI.3	SI.4	
AVERAGE	.59	.33	.76	.47	.20	.76	.28
RANGE	.40-1.00	.01-1.00	.51-.95	.27-.62	.00-1.00	.64-.87	.16-.37
STANDARD DEVIATION	.14	.20	.11	.12	.27	.05	.04
SLOPE COEFFICIENT	.24	.16	.36	.37	.13	.56	
Y-INTERCEPT	.14	.23	.01	.10	.26	-.14	
STANDARD ERROR OF ESTIMATE	.03	.03	.03	.03	.03	.04	
CORRELATION COEFFICIENT	.70	.69	.79	.85	.70	.60	

APPENDIX C

Reusability Metric Plots

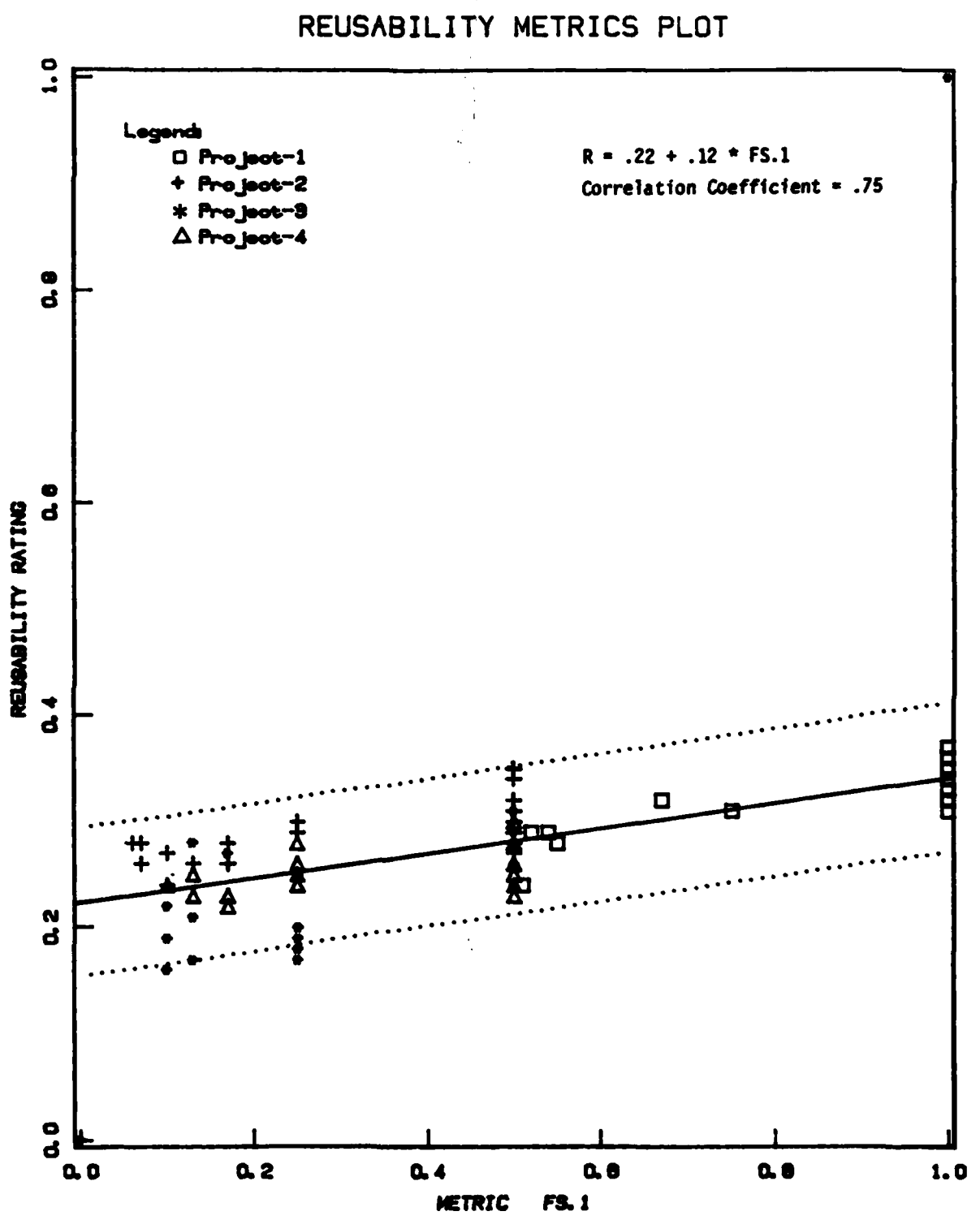


Figure C-1 : Reusability Rating vs. Metric FS.1

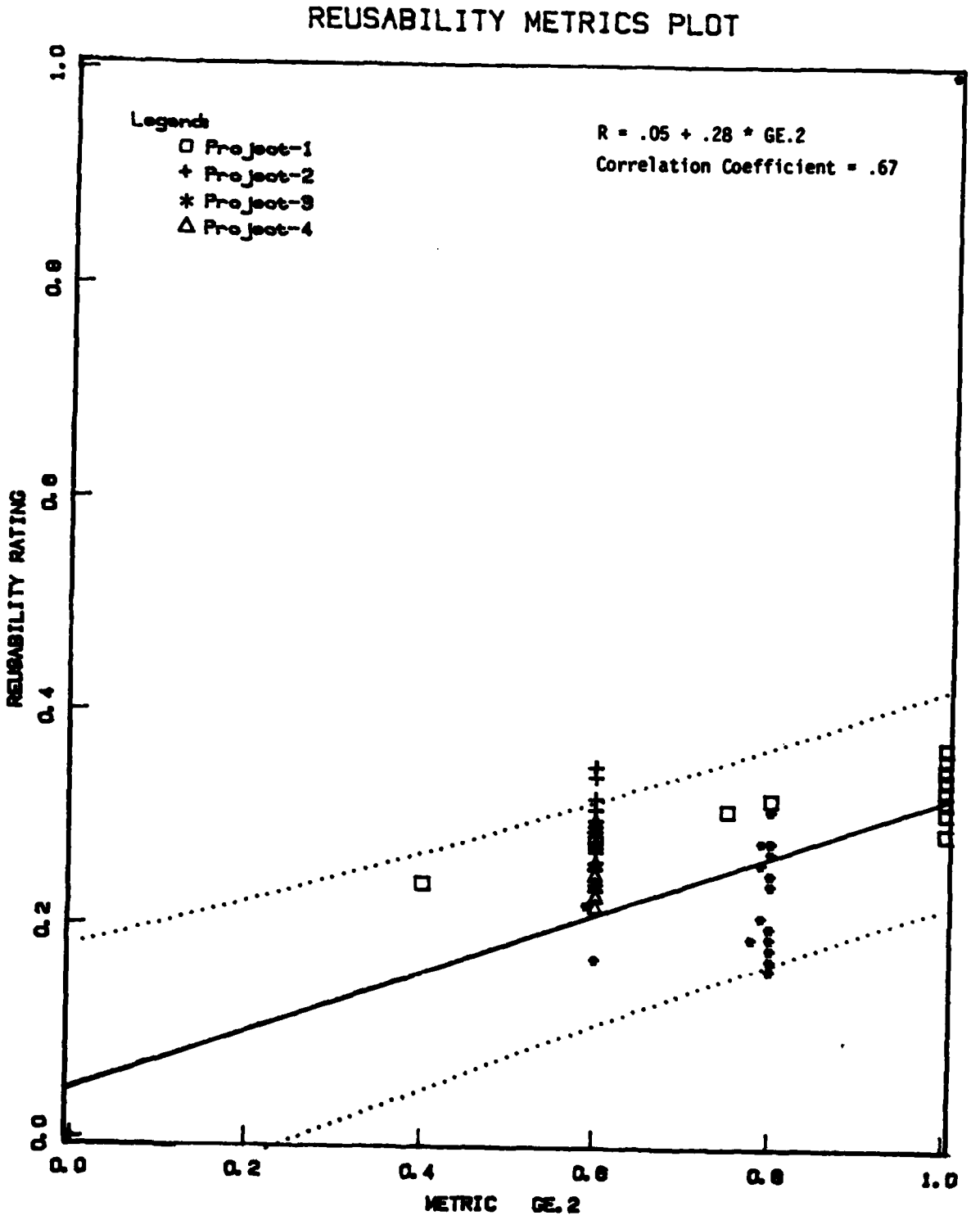


Figure C-2 : Reusability Rating vs. Metric GE.2

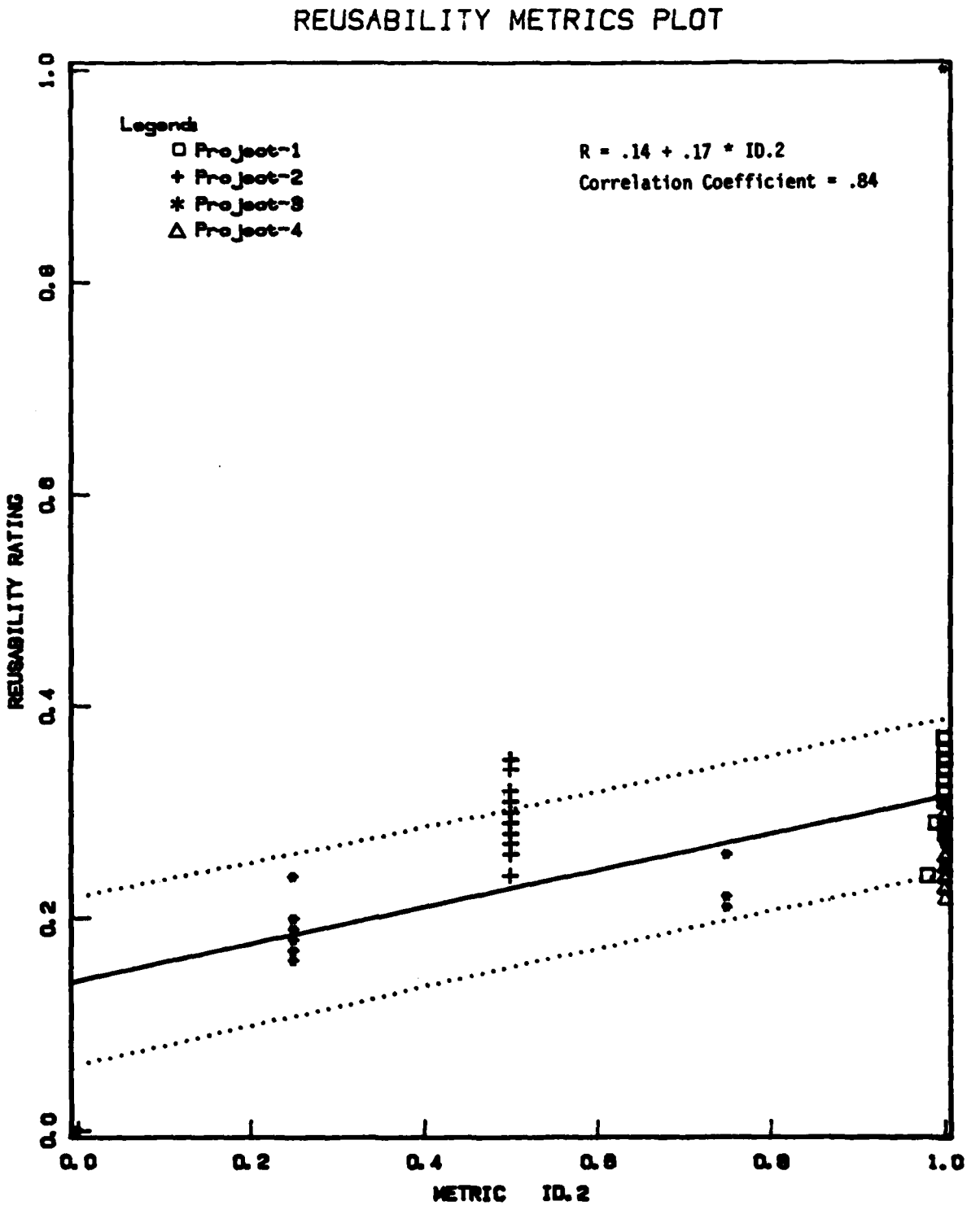


Figure C-3 : Reusability Rating vs. Metric ID.2

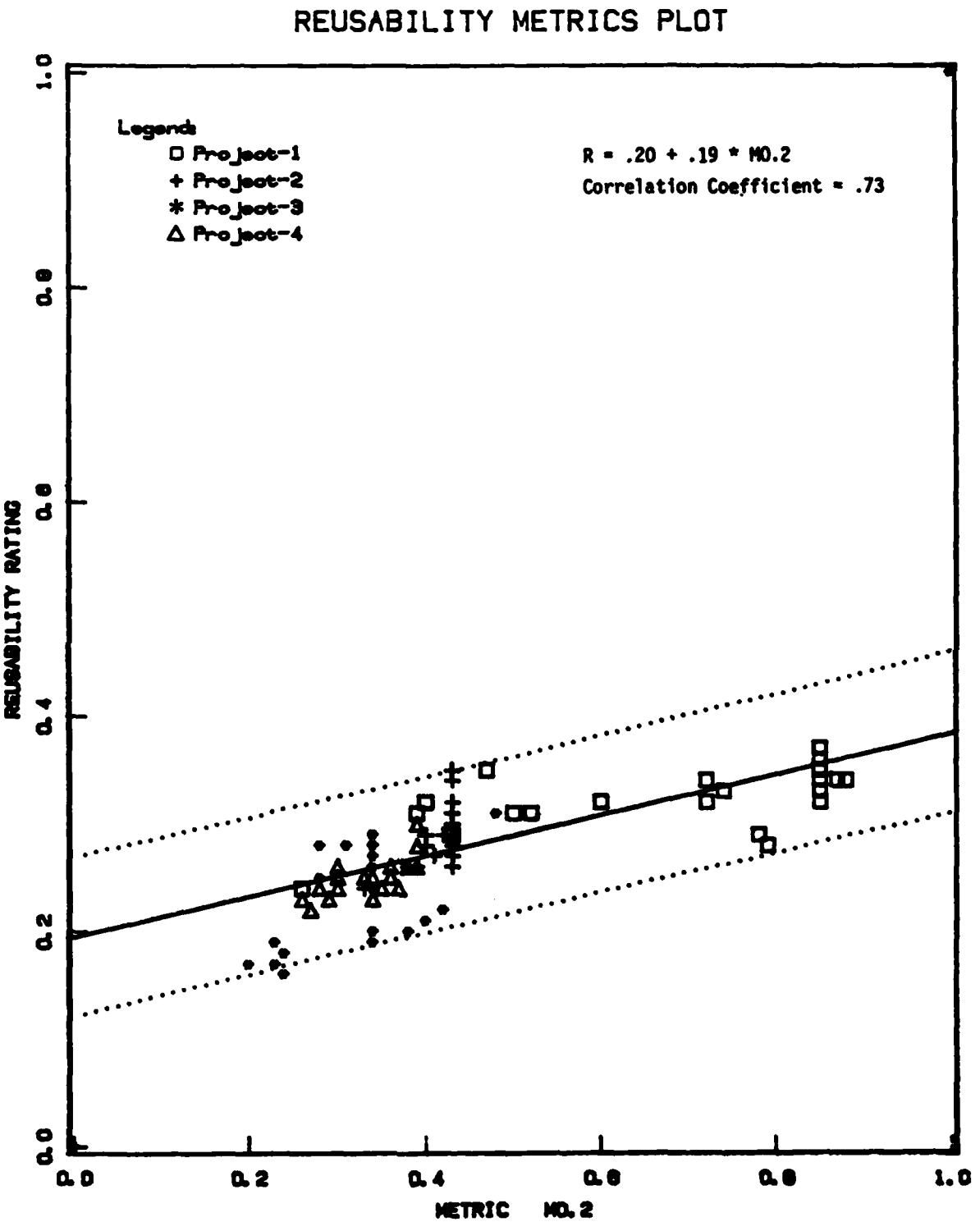


Figure C-4 : Reusability Rating vs. Metric M0.2

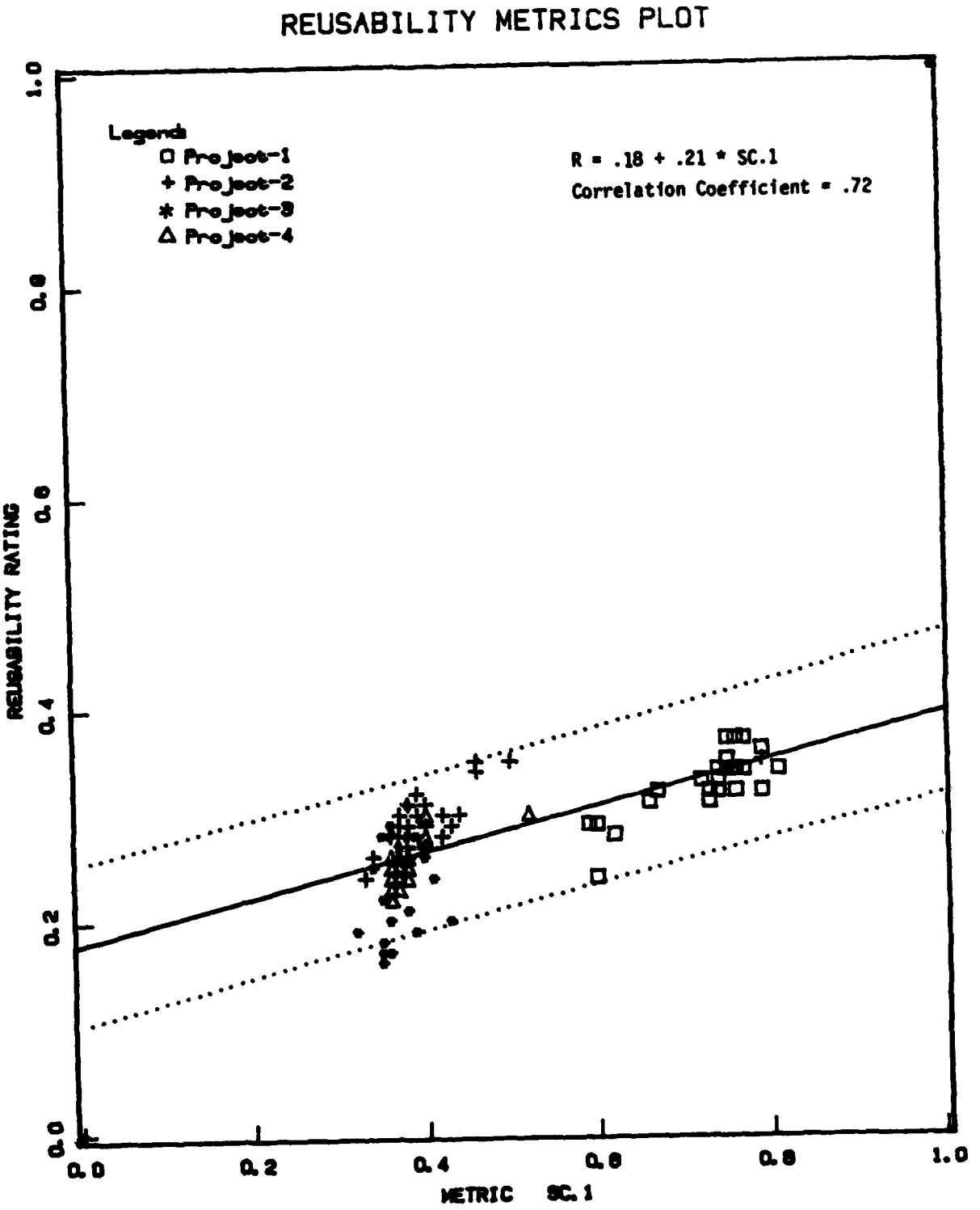


Figure C-5 : Reusability Rating vs. Metric SC.1

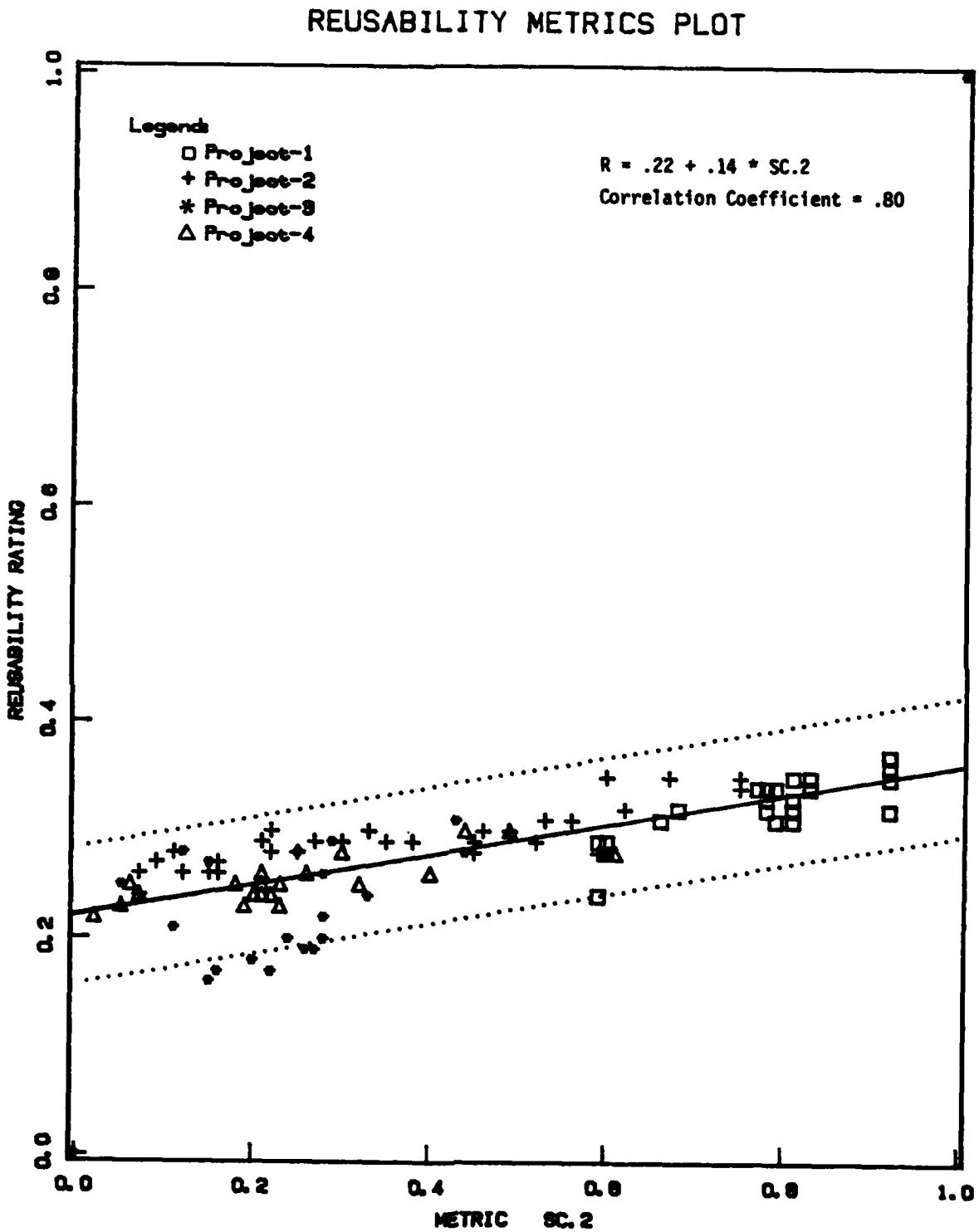


Figure C-6 : Reusability Rating vs. Metric SC.2

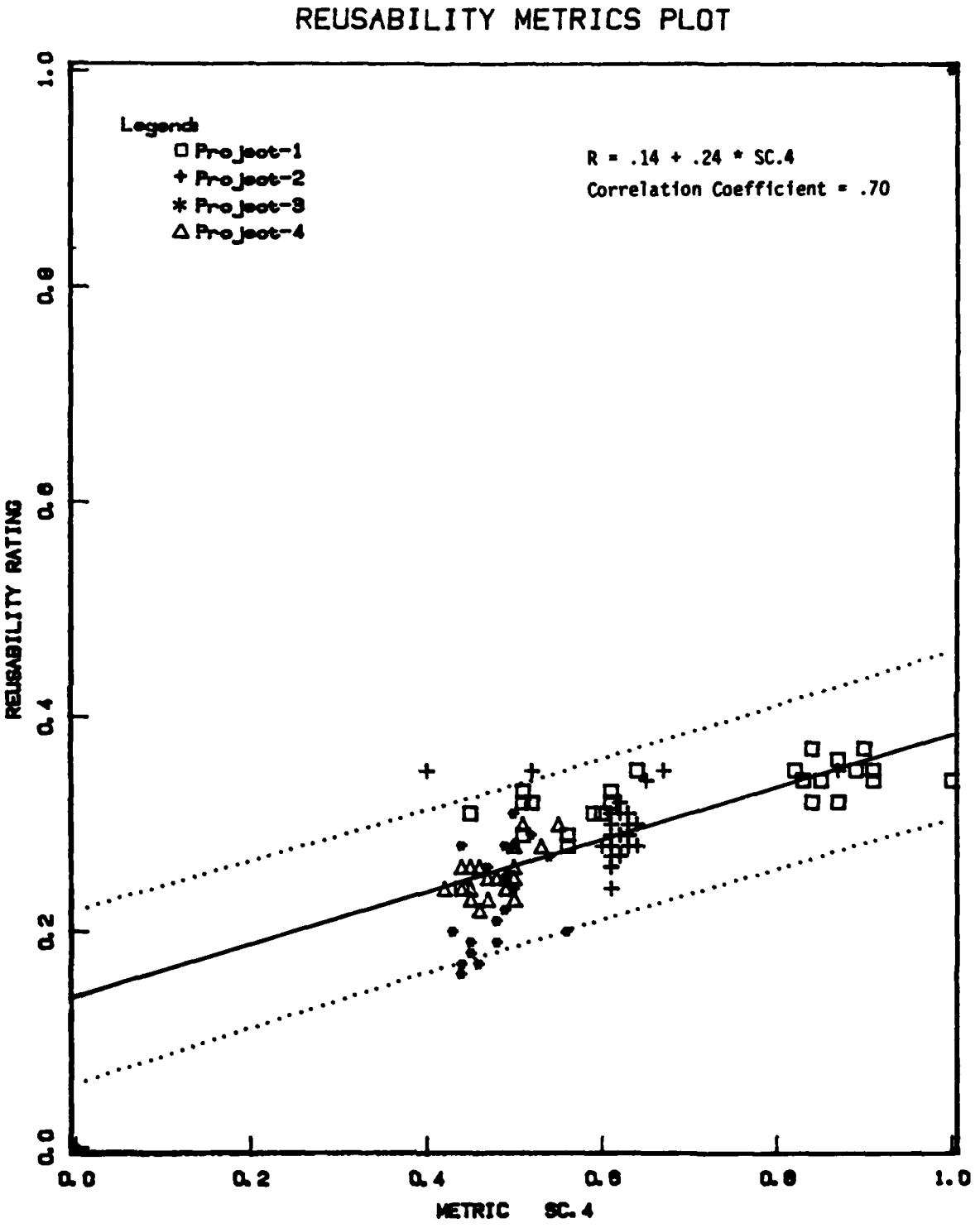


Figure C-7 : Reusability Rating vs. Metric SC.4

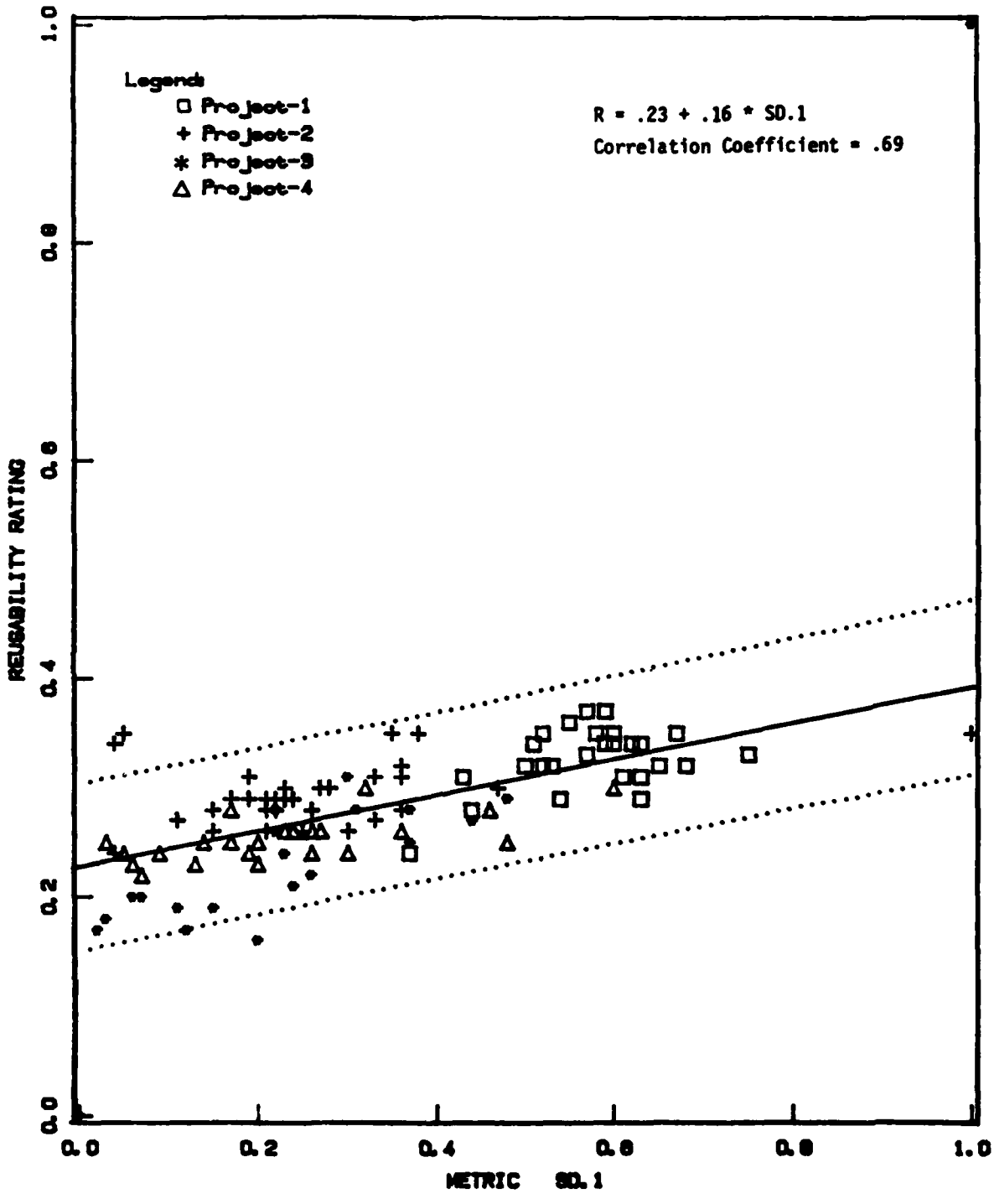


Figure C-8 : Reusability Rating vs. Metric SD.1

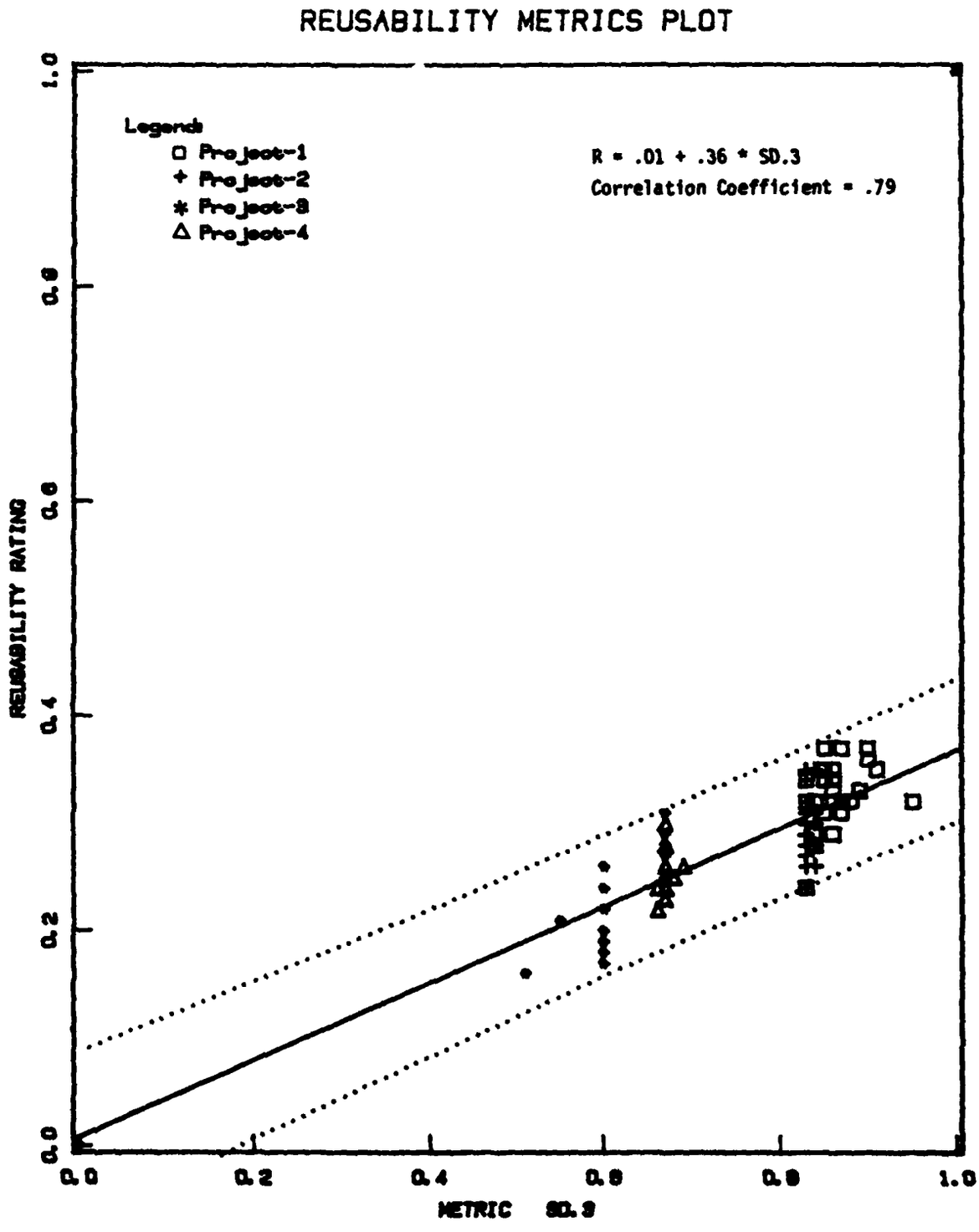


Figure C-9 : Reusability Rating vs. Metric SD.3

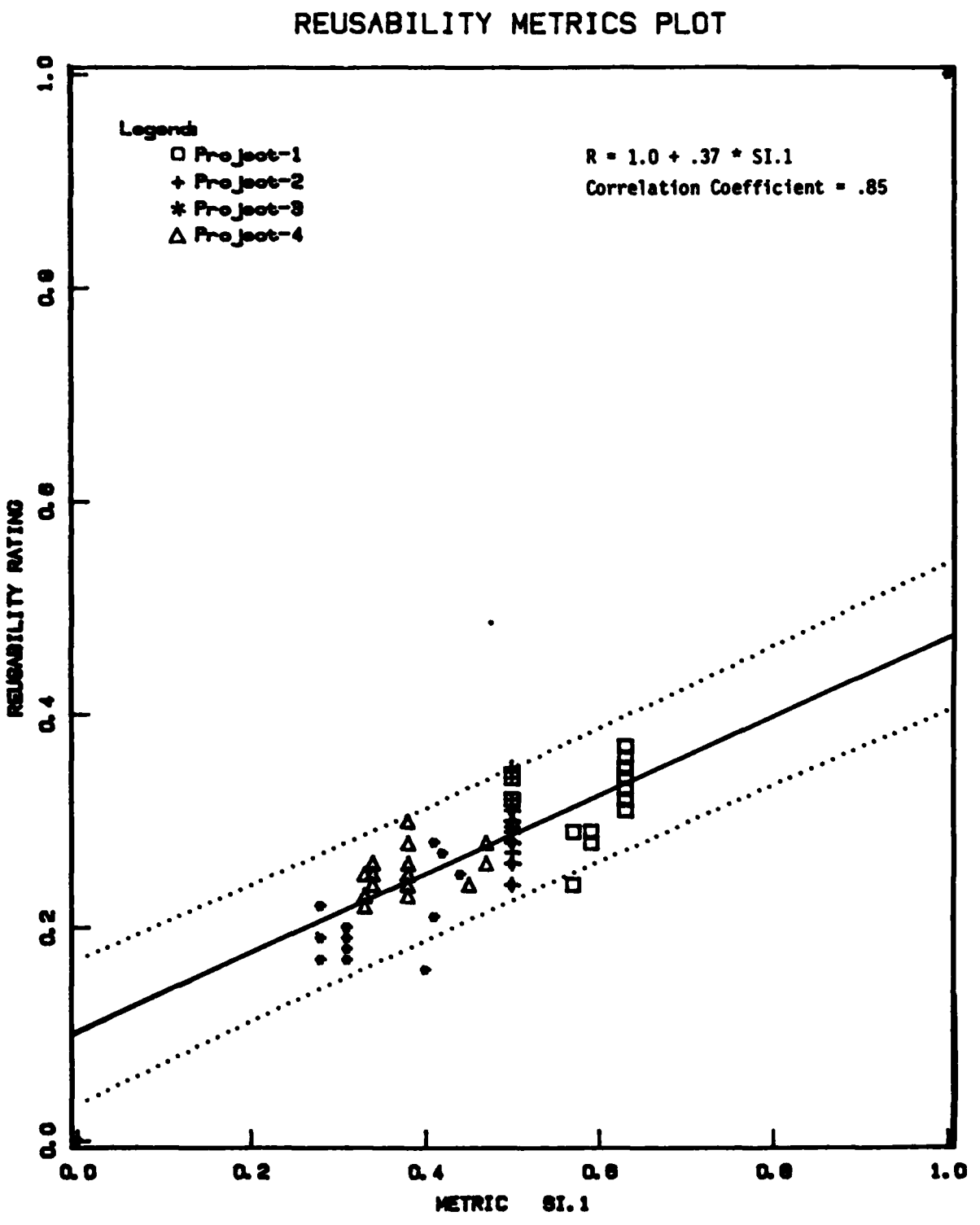


Figure C-10 : Reusability Rating vs. Metric SI.1

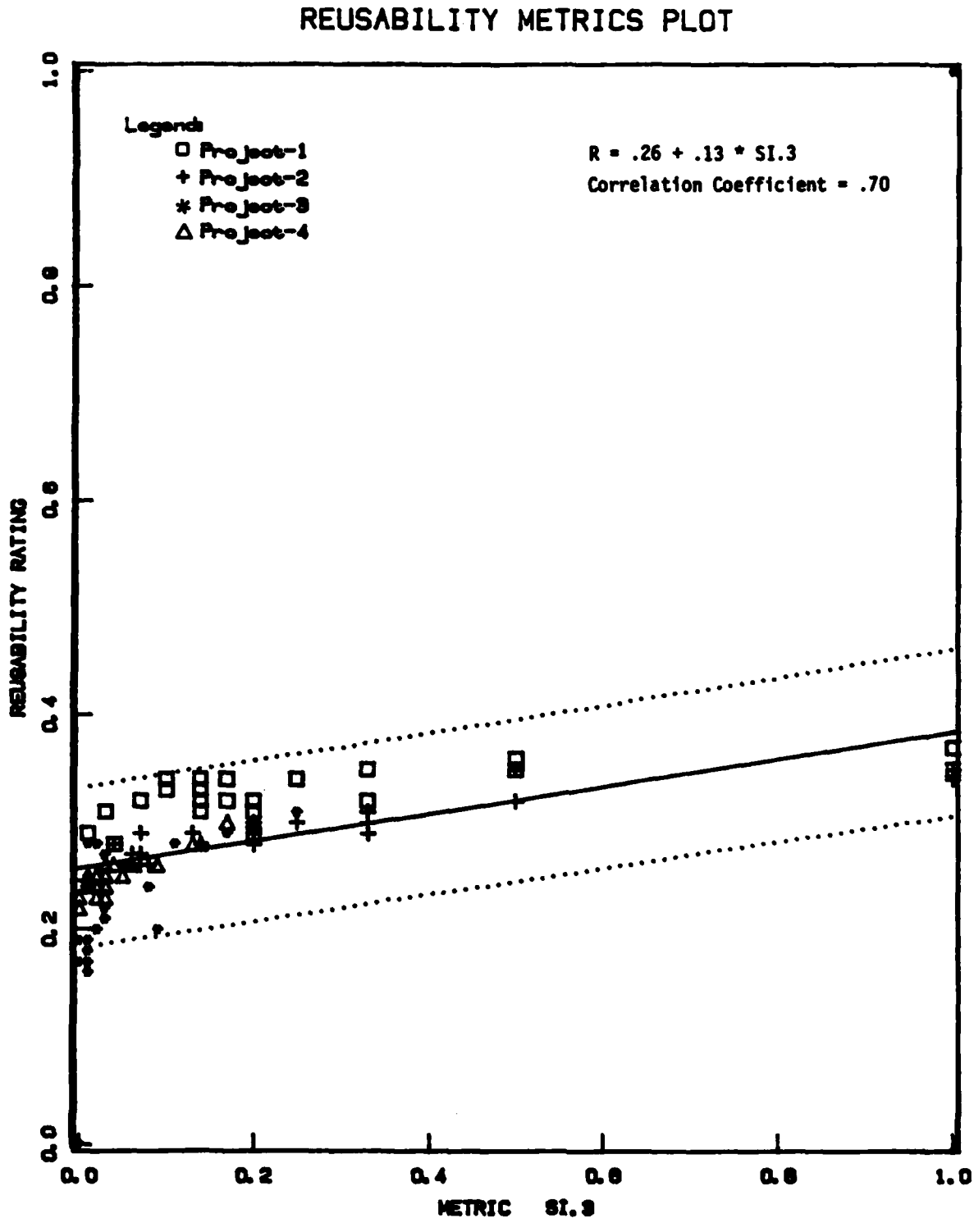


Figure C-11 : Reusability Rating vs. Metric SI.3

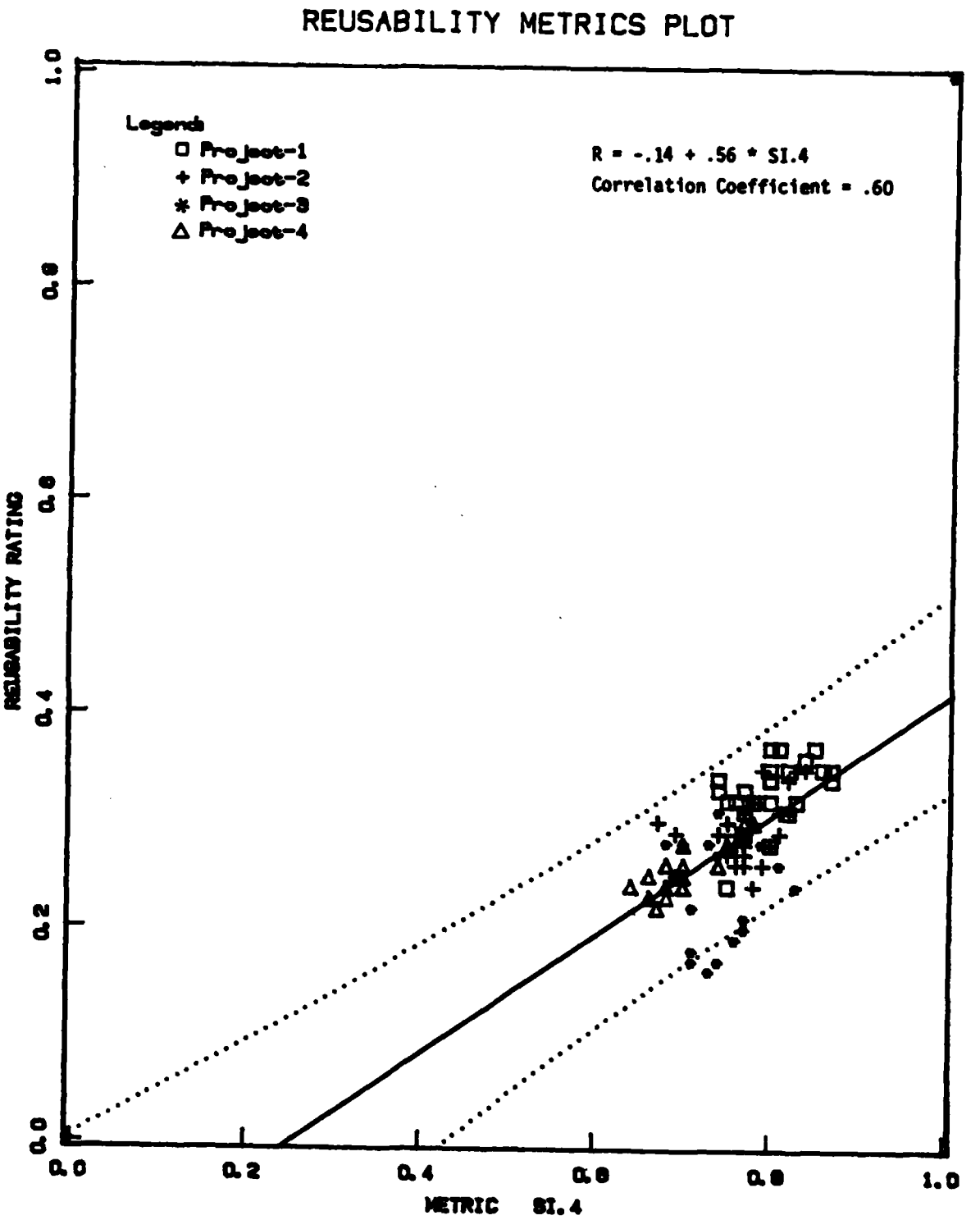


Figure C-12 : Reusability Rating vs. Metric SI.4

APPENDIX D

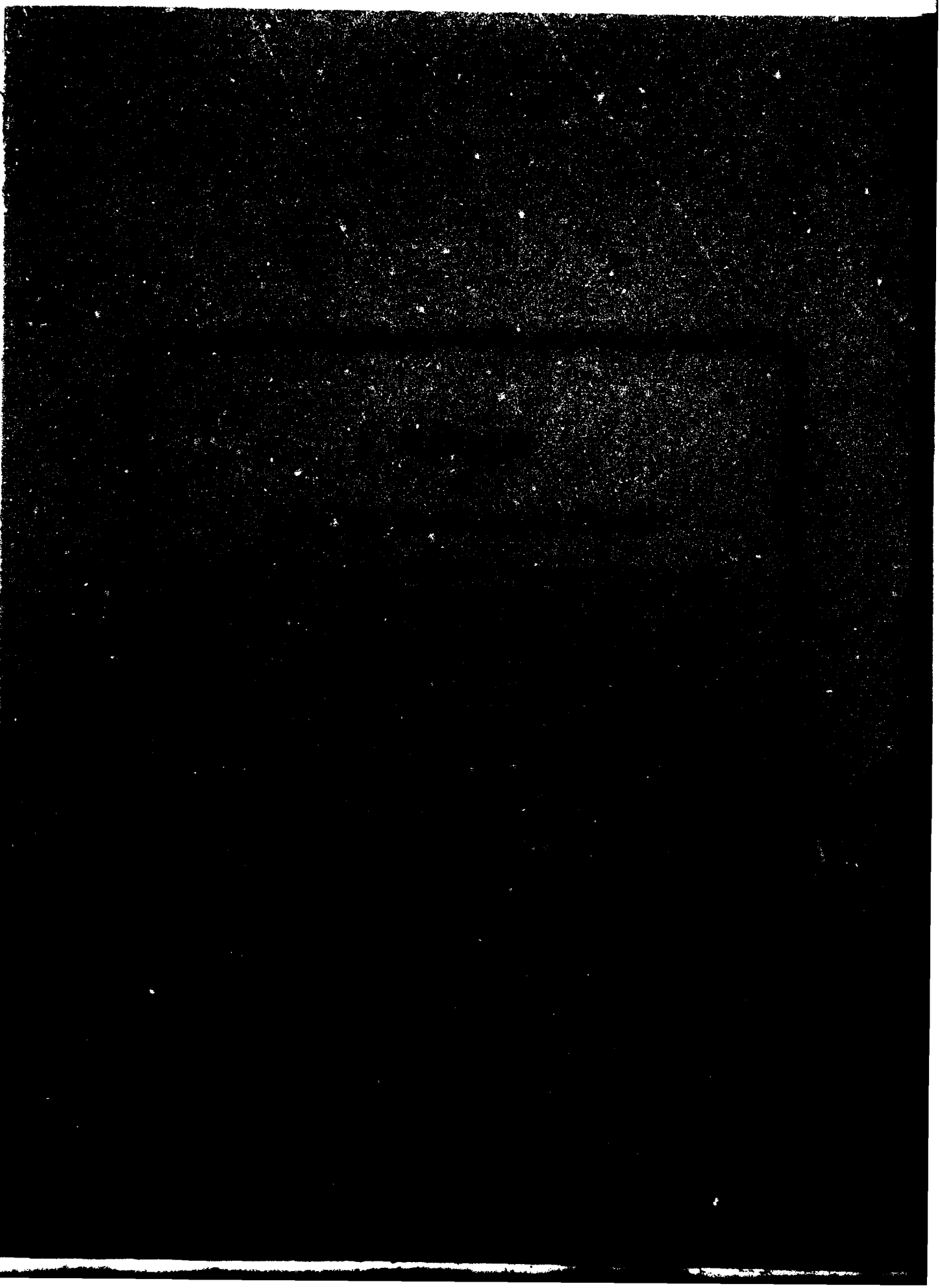
Multivariate Analysis Results

TABLE D1: Multivariate Analysis Summary

No. of Independent Variables (Metrics)	Regression Equation	Coefficient of Determination (R^2)	Standard Error
1	$.10 + .37\ M_{SI.1}$	.73	.028
2	$.13 + .29\ M_{SI.1} + .08\ M_{SI.3}$	.79	.022
3	$.10 + .08\ M_{SD.1} + .19\ M_{SD.3} + .07\ M_{SI.3}$	.84	.019
4	$.11 + .04\ M_{FS.1} + .06\ M_{SD.1} + .16\ M_{SD.3} + .07\ M_{SI.3}$	.87	.018
5	$.11 + .03\ M_{FS.1} + .04\ M_{SC.4} + .06\ M_{SD.1} + .14\ M_{SD.3} + .06\ M_{SI.3}$	.88	.017
6	$.11 + .03\ M_{FS.1} + .03\ M_{SC.4} + .05\ M_{SD.1} + .12\ M_{SD.3} + .05\ M_{SI.1} + .06\ M_{SI.3}$	.88	.017
7	$.11 + .03\ M_{FS.1} + .01\ M_{MO.2} + .03\ M_{SC.4} + .05\ M_{SD.1} + .12\ M_{SD.3} + .05\ M_{SI.1} + .06\ M_{SI.3}$	.88	.017
8	$.11 + .03\ M_{FS.1} + .01\ M_{MO.2} + .003\ M_{SC.2} + .03\ M_{SC.4} + .05\ M_{SD.1} + .12\ M_{SD.3} + .05\ M_{SI.1} + .06\ M_{SI.3}$	.88	.018

This table contains only the eight most significant n-tuple (n=1,8), with respect to the R^2 and C_p criteria.

NOTE: The variability in the rating of reusability is reduced by 88 percent when only five of the eight potential metrics are included in the equation and the standard error levels at .017. Little predictive power is gained when a 6th, 7th or 8th variable is added to the equation.




DAT
FILM